NASA Ames Research Center — QSS Group, Inc.

Mission Simulation Facility Documentation

MSF: The Missing Manual

MSF Team

December 2002

Contents

Document updated on April 7, 2005

1. Introduction

The goal of the Mission Simulation Facility (MSF) project is to provide a simulation platform for developers of autonomy software. The chosen approach uses a distributed architecture where each Federate solves a particular problem in a simulation. The various Federates are linked using a networked infrastructure based on the High Level Architecture (HLA)[3, 1]. In addition to providing basic mechanisms to exchange messages and data, HLA also supports time management and Federate synchronization.

In the HLA vocabulary, an application participating in the simulation is called a *Federate*, and the *Federation* is composed of all the Federates connected to the simulation. Federates exchange messages (interactions) and data (objects) using the Run-Time Infrastructure (RTI).

1.1. Remark

The reader of this manual should be aware that two type of classes are discussed in the text: the HLA classes, which form a hierarchy of classes of information that is shared among the Federates in a simulation, and the regular C++ classes that are used for representing the HLA classes and the simulation models.

1.2. Typographic Conventions

Italics is used for emphasis for example when new items are introduced in the text or for notes that require special attention.

HLA classes and methods and file names are rendered in monospace type: e.g. `HLAObjectClass` `federateAmbassorSet` `createBoat.cpp`

1.3. Naming Conventions

HLA-based FTK classes discussed in the text are presented in mixed-case starting with a capital letter.

Instances of HLA-based FTK classes are presented in mixed-case starting generally with a lower-case letter except for names that start with an acronym which are all in capital letters.

Class methods are represented in mixed-case starting with a lower-case letter.

1.4. Acronyms

COM	Communication entities
FOM	Federation Object Model
FTK	Federate ToolKit
HLA	High Level Architecture

LRC	Local RTI Component
MSF	Mission Simulation Facility
RFI	RTI-Federate Interface
RTI	Run-Time Infrastructure
SIM	Simulation packages
UML	Unified Modeling Language
uml2hla	UML to HLA conversion tool

2. MSF Concepts

The MSF core architecture relies on HLA mechanisms to exchange data among the various Federates of a simulation. HLA is a standard, specifying an interface of interaction in a distributed system. The HLA standard is implemented in software by several vendors through a Run Time Infrastructure (RTI) software and offers multiple mechanisms to exchange data, all based on the Publish-Subscribe scheme. The standard also describes the various rules for Federate interactions. Available RTIs implement the HLA specification, but leave the developer with several important tasks:

- Maintaining a local representation of the simulation (existing objects, data values, etc.) with the Local RTI Components (LRC)
- Ensuring data type coherency across platforms (the RTI transfers only untyped, raw data)
- Ensuring that every Federate behaves correctly in the simulation. (Each Federate needs to respond to certain request to guarantee a smooth simulation.)

To offer a consistent solution to the above problems for every Federate of the simulation, MSF provides a communication layer on top of the HLA-RTI. This layer abstracts many of the HLA internal mechanisms by providing an elaborate LRC. In addition, the MSF layer maintains a truly object-oriented approach to the communication among the Federates, thus enforcing data types and consistency. Finally, most of the rules necessary for a Federate to behave correctly in a simulation are built directly into the MSF communication layer.

The core of the MSF communication infrastructure is the Federate ToolKit (FTK), which provides the LRC, generic objects and messages, and rules for interacting within the simulation. The FTK is abstracted from any domain specific simulation and can be used for a variety of HLA simulations. The communication entities (COM) are objects and messages derived from the generic FTK classes with application specific information that form the language of a simulation for a specific domain. The COM classes are designed in a Unified Modeling Language (UML) editor and a class and code generator *uml2hla* realizes the software implementation based on the FTK (see [2] for details). Finally a Simulation Package (SIM) offers additional facilities to the MSF developer to speed up the creation of simulation components.

3. FTK

The Federate ToolKit package is designed to help MSF developers integrate their components with the High Level Architecture (HLA) framework. Details of the implementation are given in the FTK reference manual. This chapter focuses on the high level concepts implemented by FTK.

FTK roles are summarized below.

- Provide a wrapper around some fundamental methods of the the RTI (like connect, leave, etc), which offers a simplified interface to the RTI.
- Manage the class hierarchies (objects and interactions), handle retrieval, publication and subscription declarations
- Offer a Local RTI Component (LRC) which maintains a list of created and discovered objects.
- Define a Federate Ambassador implementing the RTI calls and working closely with the LRC.
- Implement the most important rules to ensure the correct behavior of the federate in regard to the rest of the simulation.

3.1. Classes representing HLA concepts

FTK defines several classes that implement the base concepts used in HLA to exchange data. The Figure 3.1 on the facing page gives an overview of the relationships of these HLA implementation classes. The mapping is defined below.

HLAInteractionClass is a class that represents an HLA Interaction

HLAInteractionMessage represents an instance of an HLA Interaction, usually referenced in FTK as a "Message"

HLAObjectClass is a class that represents an HLA Object

HLAObjectInstance represents an instance of an HLA Object, usually referenced in FTK as "Instance"

The classes depicted in Figure 3.1 on the next page offer all the necessary functionalities to handle the corresponding HLA concepts. However, these classes are generic and the user has to derive new classes from them to implement his specific classes (interactions, objclass, objects or messages). The automatic generation of these classes using the "uml2hla" tool is described in [2]. The same document presents the mapping between Communications Entities defined in UML, and the FTK based implementation classes. The naming convention for the derived classes is described in the following example of a Rover that is controlled by a DriveTo command.

msgInteraction for the class of interaction: `DriveToInteraction`

msgMessage for the interaction instance: `DriveToInstance`

objObjClass for the class of object: `RoverObjClass`

objInstance for the instance of the object: `RoverInstance`

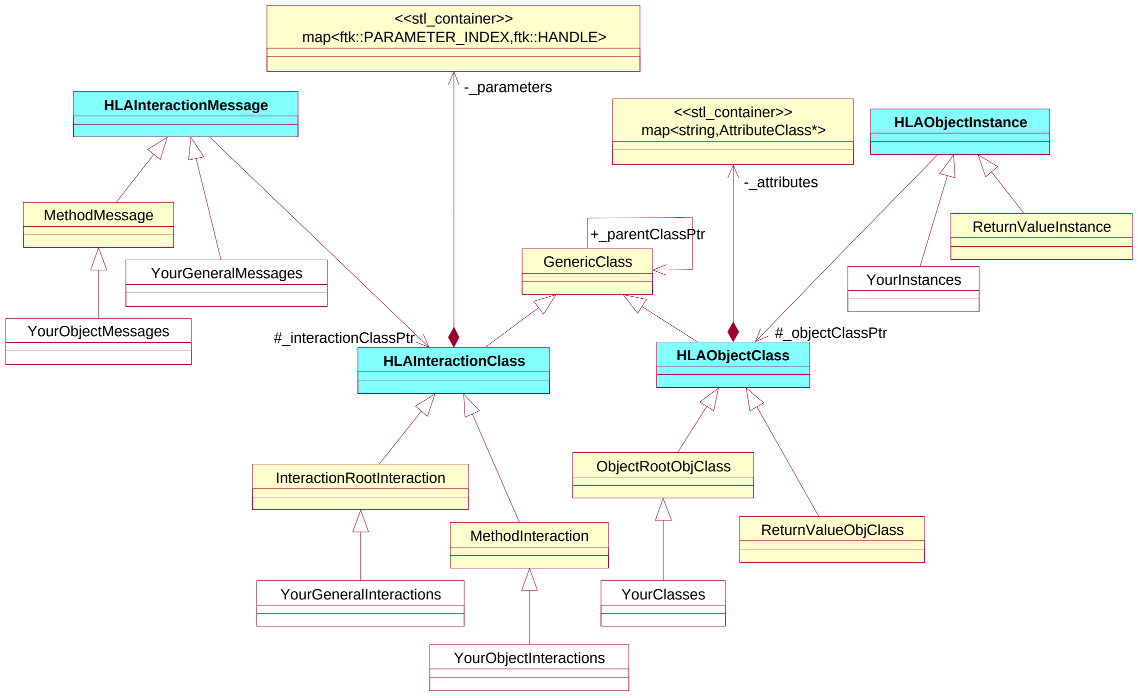


Figure 3.1.: Overview of the FTK Classes Representing HLA Concepts.

3.1.1. `HLAInteractionClass` and `HLAObjectClass`

The two FTK `HLAInteractionClass` and `HLAObjectClass` classes implement the necessary behaviors to handle the HLA concept of interaction and object classes correctly. They both derive from `GenericClass` which manages the creation of class hierarchies. Derived classes from `HLAInteractionClass` and `HLAObjectClass` are used to create C++ representation of a Federation Object Model (FOM). For each class in a FOM, a derived class from `HLAInteractionClass` and `HLAObjectClass` is defined. This allows the programmer to use class attribute qualifiers instead of string based qualifiers, as shown in the pseudo-code below.

```
// String based approach
vesselHandle = getClassHandle(``Vessel'')
speedHandle = getAttributeHandle(``Vessel'', ``speed'')
...
// Qualified approach
```

```
vesselHandle = VesselObjClass::getClassHandle();
speedHandle = VesselObjClass::getAttributeHandle(VesselObjClass::SPEED)
```

Because the C++ classes represent HLA classes, only one single instance of each should be present in a program. This is implemented by applying a singleton pattern to the class (see `SingletonHolder` class documentation). The FOM hierarchy is captured by the use of the attributes `_className` and `_parentName` of the `GenericClass` so there is not need to create an inheritance hierarchy via `HLAInteraction` and `HLAObjectClass`. In fact, all the derived classes from `HLAInteractionClass` and `HLAObjectClass` are at the same level, creating a flat structure.

The FTK LRC (RFI) is notified of the FOM hierarchy by registering to it each derived `HLAInteractionClass` and `HLAObjectClass` representing the desired FOM. The order of registration is not important, but the hierarchy tree needs to be complete, so that fully qualified HLA names can be built by the RFI[1].

The major difference between `HLAInteractionClass` and `HLAObjectClass` is that the former realizes the concept of HLA parameters, while the latter realizes the concept of HLA attributes. The `HLAInteractionClass` class maintains a simple list of parameter names and associated handles. The `HLAObjectClass` maintains a list of attributes. Attributes are represented by the `Attribute` class managing attribute names, handles, ownership and initialization state.

Following the HLA concepts of interactions and objects, the declaration of publish/subscribe intention of a Federate are done class-wise for interaction and attribute-wise for objects. An example of declaration is shown in the Sequence Diagram A.1 on page 31. The RTI handles the HLA declaration management concepts, which are captured at a higher level by FTK. The Sequence Diagram A.2 on page 32 illustrates how the declaration propagates through the Federation for Federates using FTK layer.

3.1.2. `HLAInteractionMessage`

A Federate that wants to send an HLA interaction dynamically creates an instance of a class derived from `HLAInteractionMessage`that represents the desired interaction. Each parameter can then be set individually via the accessor methods[2]. Finally, the Federate can send the message to the Federation using the `send` method. This action has an immediate effect, which means that the appropriate call to the RTI is done right away. The `HLAInteractionMessage` encodes the parameters using the XDR convention, and then builds the correct RTI based parameter set before sending the message.

Federates subscribing to a class of messages will receive messages that are appropriately derived from `HLAInteractionMessage`. In this case, the message is constructed by the FTKFederateAmbassador, which uses an object factory to create the appropriate derived class of `HLAInteractionMessage`. The parameters of the message are extracted from the RTI parameter-set and are XDR decoded. The Federate is finally notified of the new incoming message and can read its parameters using the accessor methods. An example showing the events taking place during a send/receive scenario is shown in the Sequence Diagram A.7 on page 37.

Warning: The management of the received messages is currently poorly realized by RFI. The incoming messages are simply pushed in a queue and the federate developer can pop them in order to process them. However, once a message in popped out of the queue, there is no way to re-insert (something the developer

[1]Each `HLAInteractionClass` and `HLAObjectClass` contains only its local name and its parent name.

[2]All accessor methods derived from the `HLAInteractionMessage` and the `HLAObjectInstance` classes are automatically generated by the *uml2hla* tool

could want to do if he popped the message only to have a look at it). On the other end of the spectrum, if the messages are not popped out by the federate, they will accumulate in the queue!

3.1.3. HLAObjectInstance

Each Federate can publish and subscribe to objects derived from `HLAObjectInstance`. These objects are the implementation of the corresponding FOM objects. The FOM hierarchy is directly mapped to the generated C++ objects derived from `HLAObjectInstance`, which allows a derived object to inherit from the attributes and accessor methods of its super-class.

When a Federate registers a new instance to the Federation, it owns all the attributes of the object by default. It can therefore update any attribute using the provided accessor methods. However, updating an attribute of a derived `HLAObjectInstance` does not propagate it automatically to the rest of the Federation. The notification of attribute changes to the RTI (which dispatches it to the subscribing Federates) is only done when the Federate calls the `processPendings` method of the `RFI` class. This additional step is necessary to group and optimize the updates and to avoid possible re-entrant calls[3] to the RTI, which are not supported by the implementation.

Federates subscribing to at least one attribute of a class of objects will be notified of new instances registering to the simulation. The default `FTKFederateAmbassador` handles the RTI callbacks of new discoveries by creating the appropriately derived `HLAObjectInstance` using an object factory and by adding the object to the Federate's LRC maintained by the `RFI`. The creation/discovery process is described in the Sequence Diagram A.3 on page 33. The object maintained by the RFI is a reflection of the original object registered by the creator Federate. Attributes updates are then reflected in the local object of the discovering Federate. The full process of object attribute update/reflection is described in the Sequence Diagram A.5 on page 35. This mode of data transfer can be referred to as pushing data: the creator Federate updates attributes to reflect its internal state changes, and the subscribing Federates are notified of these changes. FTK also supports a pulling mode where a subscribing Federate specifically requests an attribute update. This scenario is illustrated in Sequence Diagram A.6 on page 36. The two modes of data transfer can be mixed, although MSF simulations generally use the push scheme during a simulation run, and the pull scheme when a new Federate is connecting, to ensure that it gets all the data that have been published earlier.

If a creator Federate un-registers an instance from the Federation (it does not necessary delete it own local copy), then all Federates having discovered this object will be notified. At this point, their LRC will be updated and the object deleted (destroyed and removed from the local list). The deletion of `HLAObjectInstance` is shown in the Sequence Diagram A.4 on page 34. The developer should be extremely careful when keeping his own references to `HLAObjectInstance` since the references can disappear by the actions of a remote Federate. The only safe way to keep permanent references to `HLAObjectInstance` (beside deriving directly from the `HLAObjectInstance`) is to declare an object as a client of the desired `HLAObjectInstance` using the `InstanceClient` class. In this case the object will be notified of object deletion.

The FTK also supports exchange of ownership between Federates for each attribute of a specific object. The policy for the transaction is based on a priority scheme: only Federates with a higher priority for a class of objects can take ownership of its attributes. FTK implements the following scheme for releasing

[3] A re-entrant can arises when a Federate receives through its Federate Ambassador a request to update attributes, and tries to update them right away.

attributes: when a successful acquiring Federate releases the ownership of attributes, the ownership is automatically re-acquired by the original owner. Note that a Federate having acquired attributes will release them automatically when it leaves the Federation.

3.1.4. Remote Method Calls

In addition to Send interactions, which are anonymous by nature, FTK supports a scheme where interactions are linked to objects. This scheme allows the use of method calls on remote objects as easily as if the objects were local to the program. When defining Communication Entities using UML (see [2] for more information), the developer can define classes as having attributes and methods. The methods, which are mapped to HLA interactions, are all derived from the `MethodInteraction` class and the associated messages are derived from the `MethodMessage` class. `MethodMessage` contains two additional parameters: the handle of the destination object and the return value identifier. Carrying this information allows the FTK to implement a remote O-O method approach for interactions. The methods of `HLAObjectInstance` having a remote effect are called *commands* to differentiate them from the ones that act locally (e.g. accessor methods, class related methods).

The Federate creating an object instance is considered the modeler of the object and will therefore implement the Command method. Other Federates that have discovered that object can send commands to it. The full scenario of a remote method call is described in the Sequence Diagram A.8 on page 38. Because of the networked nature of a distributed MSF simulation, the call to a command method is not blocking – it simply returns a command identifier that can later be used to query the status of the command. The status of a command is implemented by creating “`ReturnValue`” objects as shown in Sequence Diagram A.9 on page 39.

Warning: the current implementation of FTK does not clean up after the return values. A good scheme for deciding when a return value object is no longer useful has to be put in place in order to delete the “old” return values and avoid an explosion of instances in the simulation.

3.2. LRC and Federate Ambassador

The communication of a Federate with its Federation is realized with two interface classes: the RTI Ambassador, which allows the Federate to control the Federation execution, and the Federate Ambassador, which gives the Federation execution access to the Federate. Both classes are handled by the FTK RFI as shown in Figure 3.2 on the next page. The RTI Ambassador is created by the RTI implementation, but it is up to the developer of the Federate to implement the Federate Ambassador. FTK provides a Federate Ambassador class (`FTKFederateAmbassador`) that includes all the callbacks used in an MSF simulation. The Federate Ambassador works closely with the RFI as it acts on the LRC to process new messages, object attributes etc. Together, the `RFI` and the `FTKFederateAmbassador` classes relieve the developer from implementing any details regarding the interaction of the Federate with the Federation.

Another benefit of using the FTK library is that all the rules of HLA are already built into the library, so the developer does not have to worry about the many complex behaviors. The HLA rules implemented within the RFI, the FTKFederateAmbassador and the HLA classes include among many others:

- Correct declaration propagation and notification
- Reply to attribute requests from other federates

- Provide consistency with object registration / deletion
- Manage ownership transactions

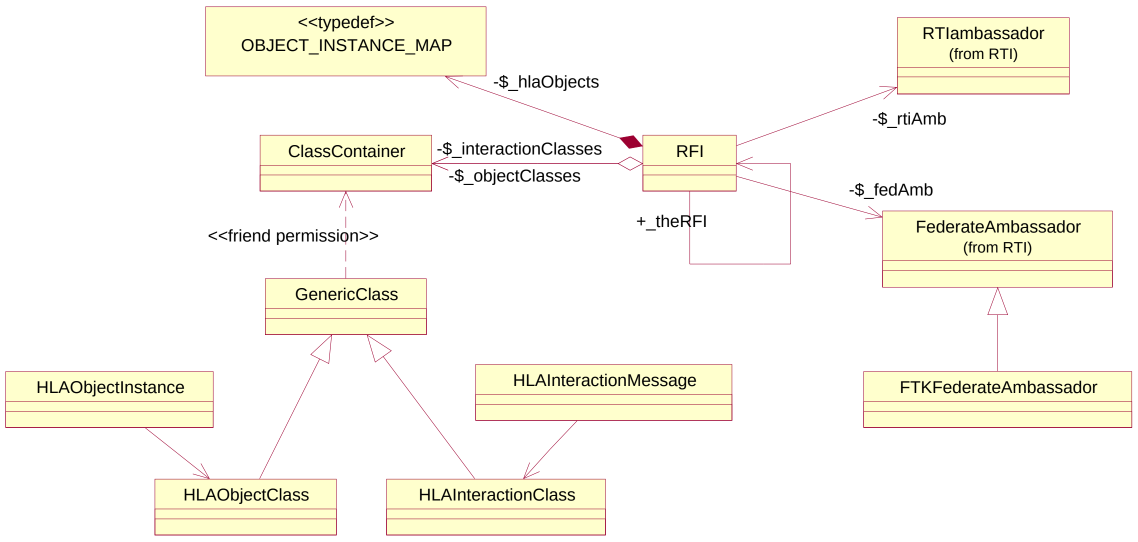


Figure 3.2.: Implementation of the LRC in FTK.

We can see in Figure 3.2 that the `RFI` has a reference to the `RTIambassador` as well as one to a `FederateAmbassador`. At `RFI` initialization, it is necessary to provide the RFI with an implementation of the FederateAmbassador. This is normally done using the provided `FTKFederateAmbassador`, but the developer can also derive his own Federate Ambassador to meet specific purposes.

The `RFI` maintains two class hierarchies, one each for interactions and object classes. This allows the `RFI` to build the fully HLA-qualified names from the parent names and class names, retrieve the handles for classes, parameters, and attributes, and to do other tasks necessary for the the correct behavior of the simulation.

The `RFI` maintains the LRC, which is mainly implemented with the `OBJECT_INSTANCE_MAP` container. Parallel containers are used to provide rapid access to the objects regarding their names or handles.

Note: The LRC is currently integrated in the `RFI` *class. However, we plan to make it a separate class to effect a better distribution of responsibilities and to create a cleaner interface.*

3.3. Helper classes

All classes composing the FTK library are described in the reference manual but some of them are briefly presented here.

4. COM

4.1. Overview

The COM package contains classes that are used for data communication among participating Federates in an MSF Simulation. The design, creation, and code generation of these classes is described in the *uml2hla* documentation. This chapter describes the concept and hierarchy of the classes that are included with this MSF release.

4.2. Hierarchy

At the highest level, the library contains general entities, such as rover, locomotor, manipulator, and terrain to support the current simulation domain, which is focused on planetary surface rover missions. At a lower level, the user is supplied with classes that communicate data and commands related to devices. The COM library of objects allows a user to create a variety of simulations without having to design and create his own communication classes. The library will grow as users create different or specialized simulations.

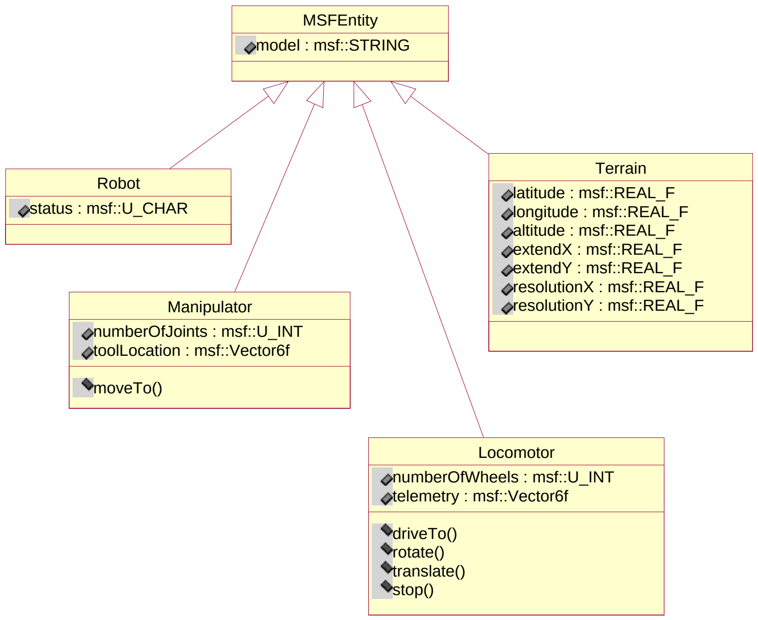


Figure 4.1.: Overview of General Entities.

The COM classes are divided into three major packages: General Entities, Devices, and Properties. The separation of the properties from the devices allows modeling a scenario at different levels of fidelity and requires the user to represent only the objects and data that are exchanged among the participating Federates. For example, in a rover simulation that does not include a power model, it is not necessary for devices, such as motors and instruments, to model the power consumption. It is therefore not required to attach the Electric property to these devices, since no other Federate would subscribe to it.

The Devices package contains actuator, sensor, detector, instrument, and battery as shown in Figure 4.2, all of which are found in a typical rover engaged in surface exploration. In the near future, as MSF is being used for building simulations and executing a variety of scenarios, more specialized equipment may be added to the library. It is our goal to provide a library that contains all devices that are most often used in simulations in the current domain to support those users of MSF who do not want to learn the details of creating their own communication classes.

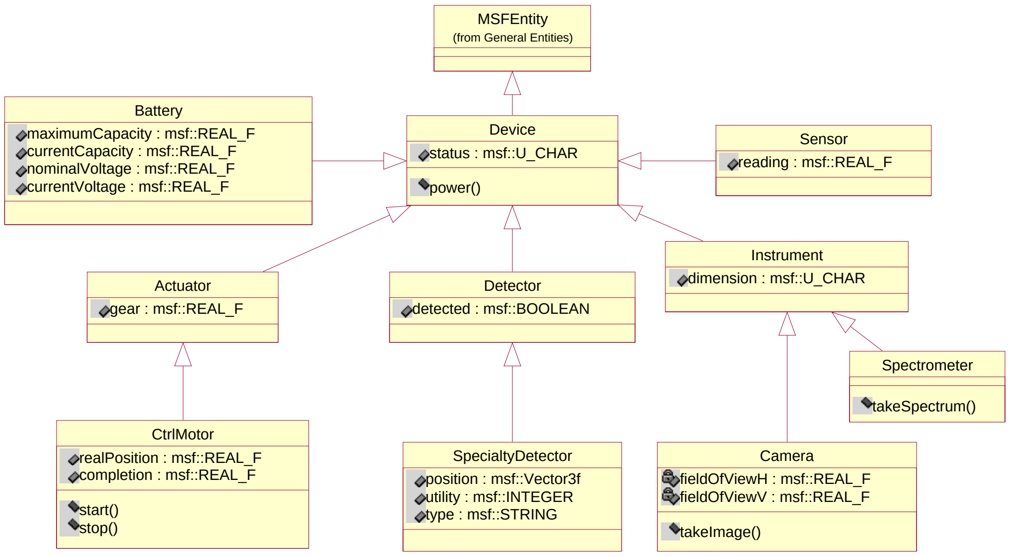


Figure 4.2.: Overview of Devices.

The third package contains properties that can be attached to any communication entity. Entities that don't have properties attached to them are only represented as concepts in a simulation. For example, a trajectory generator may not have a physical representation in a simulation so there is no need to represent that information in the entity. On the other hand, a camera may be visualized in a viewer and its mass may be considered by the dynamics engine to compute the rover's pose. If the user wants to model the camera's power consumption, he can further enhance the camera model by attaching the Electrical property to it.

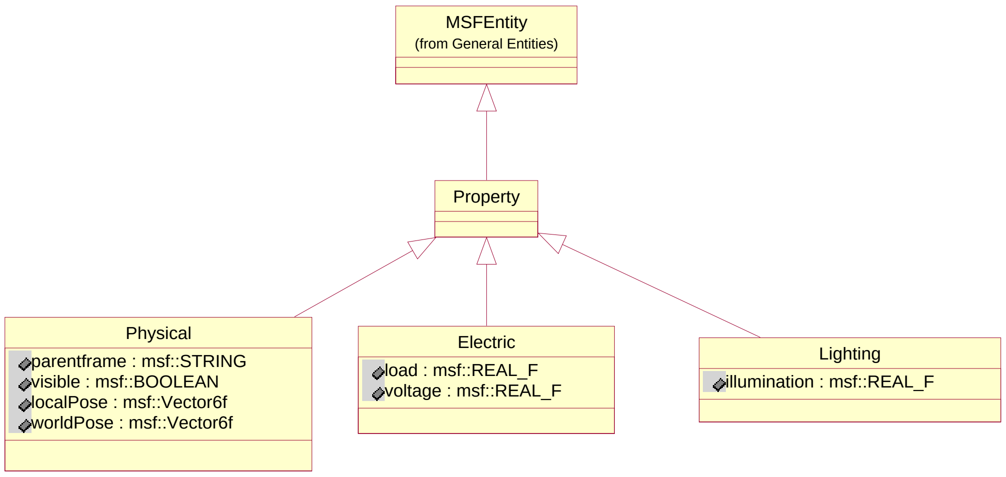


Figure 4.3.: Overview of Properties.

5. How to build an MSF Federate

Figure 5.1 shows the main phases in the lifetime of an MSF Federate:

- Initialize the RTI-Federate Interface (RFI):
  - Define the Federation name and Federation File to be used
  - Specify the Federate Ambassador to be used
  - Declare the Federation class structure
- Connect to the simulation
- Retrieve all the handles that were dynamically defined by the RTI
- Declare what data the Federate will publish and to what data it will subscribe
- Start the simulation loop
- Terminate at the end of the simulation or after the simulation loop returns.

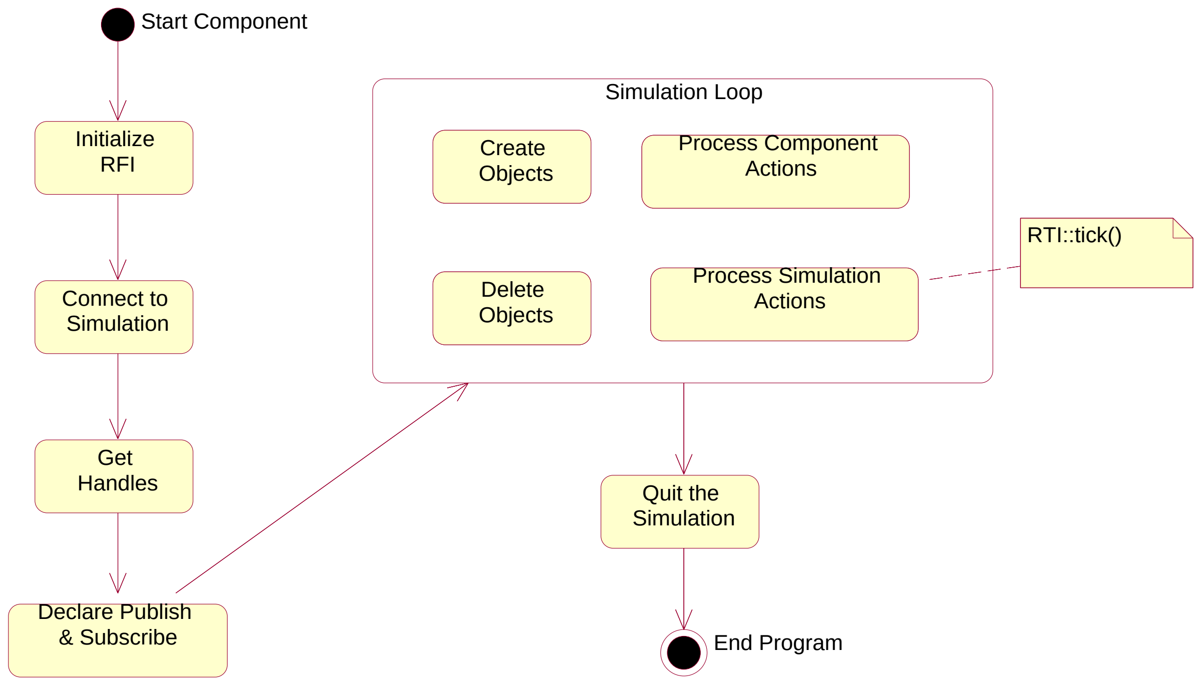


Figure 5.1.: Overview of MSF Component Activities.

Each MSF Federate is responsible for modeling a particular subset of the entire simulation. In the simulation loop a Federate generates its new states according to external inputs and then communicates the new states to other Federates in the simulation. Each Federate also services any requests that it may have received from other Federates in the simulation. Some Federates (e.g. a data logger) participate only passively in a simulation. These type of Federates consume data but do not produce information that needs to be shared with other Federates. However, the data they produce is useful for other actors, such as a 3D viewer displaying information to a human user.

The creation and deletion of objects can take place at any time in the simulation loop, or even before the simulation loop begins. The only prerequisite for creating objects or sending messages is that the Federate declare to the RTI the classes of the objects it wants to publish and the classes of the messages it wants to send. Before a Federate can declare publications, the class handles need to be retrieved.

It should be noted that it is possible to change the Federate's Publications/Subscriptions at any time[1]. The class structure can also be extended after the normal initialization phase, but then the Federates have to retrieve the class handles by querying the RTI after any addition in the class structure. The RFI interface allows dynamic modification of the class structure, but this method is not recommended since the reference to the the class structure, from the RTI's point of view, is the Federation file, which is static during a simulation run. In future versions of MSF, the declaration of the class structure will be automated[2], or embedded in an application library.

5.1. Overview of Examples

To illustrate the creation of MSF Federates, several simple examples will be studied. The full source code for these examples can be found in Appendix B on page 40. Because MSF uses a distributed scheme, at least two Federates are necessary to show the communication across the simulation. Therefore, almost all the examples come in pairs of two Federates.

A first scenario uses the `createBoat.cpp` and `listenBoat.cpp` programs. The first program simply creates a new Sailboat instance, and then updates its attributes: heading, speed and heel. The second program discovers any Sailboat in the simulation and prints its current attributes.

A second scenario uses the same object class in two others programs: `modelBoat.cpp` and `controlBoat.cpp`. The `modelBoat.cpp` file describes a model of a boat that does nothing except print the commands it receives from another boat, in this case the commands issued by the program described in `controlBoat.cpp`.

The two following sections (Section 5.2 on the facing page and Section 5.3 on page 21) are based on the `createBoat.cpp` program. The next section (Section 5.4 on page 22) uses code samples coming from the `listenBoat.cpp` program. Finally, Section 5.5 on page 24 uses the `modelBoat.cpp` and `controlBoat.cpp` programs.

[1]This is implemented but requires more testing before it should be used.

[2]Because we chose a strongly "typed" data structure for representing the MSF Federates, the class structure is embedded into the class names of several classes including the Object, Interaction, and Attribute classes. It is therefore not possible to build the class structure at run-time by reading it from the Federation file. The chosen approach detects naming problems at compile time.

5.2. Initializing an MSF component

The code in listing 5.1 begins with the statement declaring an RFI (Run-Time-Infrastructure – Federate – Interface). Only one RFI can exist per Federate and the RFI can be accessed from anywhere in the code by static methods. The RFI maintains an LRC and will do the necessary calls to the RTI regarding the Federate request. To allow the RTI to execute requests to the Federate, a Federate Ambassador needs to be implemented, which is done on the next line. In most cases, an MSF Federate developer can use the default MSF Federate Ambassador (`FTKFederateAmbassador`). The `FTKFederateAmbassador` implements the basic Federate Ambassador methods used in an MSF simulation. If the Federate developer has a specific need, he can derive the FTKFederateAmbassador to implement additional methods, or change the behavior of already implemented methods. The Federate Ambassador is assigned to the RFI with `federateAmbassadorSet`. This step is necessary, so that the RFI can inform the RTI which Federate Ambassador to use during the connection sequence. In addition, the Federate declares the name of the Federation it wants to join and the filename of the Federation file (FED file) it relies on with the commands (`FederationNameSet` and `federationFileSet`).

```
RFI theRFI;
FTKFederateAmbassador* theFA = new FTKFederateAmbassador();
theRFI.federateAmbassadorSet(theFA);

theRFI.federationNameSet("MSF-TEST");
theRFI.federationFileSet("navigation.fed");

HLAObjectClass& vesselClass = SingletonHolder<VesselObjClass>::Instance();
HLAObjectClass& sailingBoatClass = SingletonHolder<SailboatObjClass>::Instance();
theRFI.registerObjectClass(vesselClass);
theRFI.registerObjectClass(sailingBoatClass);
```

Listing 5.1: Initialization

The next step in the Federate initialization is to declare the Class structure. Although a standard HLA simulation uses a Federation File that can be read at runtime, MSF uses or more strongly “typed” approach, and therefore every Object or Interaction Class that will be used in the simulation (and defined in the Federation File) is declared as a class (see the *uml2hla* documentation). The simple example described here uses the same class ctructure as the one presented in the *uml2hla* documentation and is represented in Figure 5.2 on the next page. The example program only creates objects of class Sailboat, but it is also necessary to declare the parent classes of Sailboat to ensure that the RFI can build a fully qualified HLA name[3] for each class. References to classes composing the class structure are defined using the Singleton pattern. These classes are registered to the RFI, which builds the class structure from it. Note that the order of registration is not important (VesselClass could have been registered after SailboatClass), but registering all the parent classes of the classes used by the Federate is critical (e.g. VesselObjClass needs to be registered since SailboatObjClass is a child of it).

5.2.1. Connecting to the Simulation

After the necessary initialization, the MSF Federate can be connected to the simulation with the unique call `joinFederation`. Since multiple Federations (simulations) can concurrently exist in an HLA simula-

[3]In the simulation, the fully qualified name of Sailboat will be `ObjectRoot.Vessel.Sailboat`

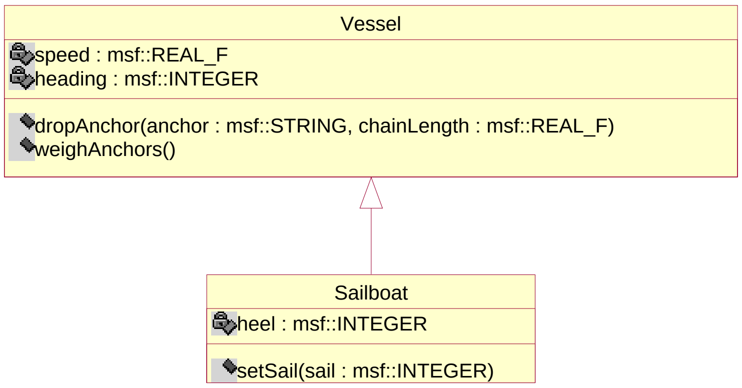


Figure 5.2.: Class Structure.

tion, the MSF Federate connects to the particular Federation execution that has been named with the call, `FederationNameSet`. The method `joinFederation` first checks if a Federation execution with the given name already exists. If it does already exist, the Federate simply joins it, otherwise the RFI instructs the RTI to create a Federation execution with the given name and then joins the newly created Federation execution.

```
int wait = Output::wait("Trying␣to␣Join␣the␣Federation");
int error =  theRFI.joinFederation();
if ( error ) {
   Output::fatal("An␣Error␣has␣occured␣when␣joining:␣stop␣the␣program!\n");
}
else {
   Output::ready(wait);
}
```

Listing 5.2: Connection

5.2.2. Declaring the Federate's Intention

After the MSF Federate has connected to the Federation execution (simulation) without errors, it can declare its intentions: what classes of objects and interactions[4] the Federate will produce and to what classes the Federate wants to subscribe. In fact, HLA allows the publication and subscription to be defined at a finer level for object classes since the publish/subscribe status of every object attribute can be individually set. For this reason, each FTK `HLAObjectClass` maintains the publish and subscribe status of each attribute (class wise, not instance wise). By default, every `HLAObjectClass` publishes and subscribes to all attributes but the status can be set globally with `publishAll`, `subscribeAll`, `unPublishAll` and `unSubscribeAll`. The publish/subscribe status can also be set for each attribute individually with `setPublish` and `setSubscribe`.

```
wait = Output::wait("Getting␣all␣handles");
theRFI.getHandles();
```

[4]Interactions will be treated later in this chapter

```
    Output::ready(wait);

    wait = Output::wait("Publishing␣federates␣classes");
    sailingBoatClass.publishAll();
    sailingBoatClass.unSubscribeAll();
    sailingBoatClass.PublishSubscribe();
    Output::ready(wait);
```

Listing 5.3: Declaring Publications and Subscriptions

When the correct publish/subscribe policy for each object class is obtained, the MSF Federate can declare its intention to the RTI (and thus to the rest of the simulation). This declaration is achieved by calling the `PublishSubscribe` method of the corresponding `HLAObjectClass`. Only at this point is the RTI notified of the Federate's intention. The other methods for setting the publish/subscribe states of classes have no direct effect on the RTI: they only change the local state of these classes. Therefore, if the publish/-subscribe policy of a class is changed later in the program, then the `PublishSubscribe` method has to be called again to notify the RTI of the changes. It should also be noted that although the MSF developer has to register the full class structure to the RFI to completely define the full class names, he only needs to declare publication to the RTI for the classes of objects that the Federate will produce (i.e. in the present example, it is not necessary to declare `VesselClass.PublishSubscribe` since this Federate will never create objects of class `VesselObjClass`).

5.3. Producing objects and updating their attributes

If an MSF Federate publishes a given object class, it can create instances of this class. As shown in Listing 5.4, the first step is to create the instance (new `SailboatInstance`) and the second step is to register the instance to the RFI (`registerObjectInstance`). Only this latter call actually notifies the RTI (and thus the rest of the simulation) of the creation of a new instance in the simulation. Once an object instance has been registered to the RFI, the developer transfers the ownership to the RFI. Although the object has been allocated dynamically, it should not be deleted with the `delete` operator! In fact, once an object has been registered to the RTI, another Federate may take ownership of it and even delete it. Therefore, the user has to let the RFI manage the registered objects. The RFI will delete the registered objects at program termination, or remove them earlier if another Federate with the right privilege is requesting it. However, the MSF developer can call `removeObjectInstance` if he wants to remove the object from the Federation and from the RFI lists[5]. Finally, it should be noted that the object instance's name can be individually set by the call, (`nameSet`). If the developer does not set the name of the object before registering it, a name will be automatically assigned to the instance (the RFI makes the appropriate call to let the RTI define the name). Once the name has been assigned to an instance, it is valid throughout the lifetime of the object. The name of each instance has to be unique among all the objects in the simulation. This is guaranteed when the RTI assigns a name automatically, but if the user tries to register an object with a name that already exist, the registration will fail.

```
        wait = Output::wait("Creating␣one␣Sailboat");
```

[5]There is another call: `unRegisterObjectInstance`, but it should be used only by the `HLAObjectInstance` itself. It will be interesting see what happens if a user uses this call, which in this case should return the object itself and allow the user to delete the object. Finally, `removeObjectInstance` should NOT delete the object from the RFI lists, (as it does now) if the object is not owned: this is a bug in the current implementation since it corrupts the LRC!

```
    SailboatInstance* boat = new SailboatInstance();
    boat->nameSet("my_best_boat");
    theRFI.registerObjectInstance(boat);
    Output::ready(wait);
```

Listing 5.4: Creating One Object

The simulation loop of the sample program presented in this section simply updates the attributes of the sailboat. This is shown in Listing 5.5: the `attributeSet` methods are changing the local values of the object instance, but these changes are not propagated to the simulation. Only when the Federate calls `processPendings` will the RFI notify the RTI of all the new states of its local objects. It is very important to understand that all the calls done on `HLAObjectInstance` are only changing the LRC maintained by the RFI and only when calling `processPendings` will the other Federates in the simulation receive the updates with the new states from the RTI.

```
    for (int n=0; n<100; n++) {
        boat->headingSet(270);
        boat->speedSet(n%20);
        boat->heelSet((int)((n-50)/10));
        RFI::processPendings();
        msSleep(1000);
    }
```

Listing 5.5: Simulation Loop

5.4. Discovering objects and reading their attributes

Federates that are interested in a specific class of objects have to subscribe at least to one of the class attributes in order to be notified when a new object instance is created. This is shown in Listing 5.6 where the Federate declares its intention to subscribe to all attributes of Sailboat. The next step is to instruct the RFI what to do when objects of class Sailboat are discovered. The RFI uses an object factory to create object instances of a specific type without having prior knowledge of these classes. The `registerCreateInstance` method tells the RFI to associate the function call (`createSailboat` in our case) with a class of objects.

```
  wait = Output::wait("Publishing␣federates␣classes");
  sailingBoatClass.unPublishAll();
  sailingBoatClass.subscribeAll();
  sailingBoatClass.PublishSubscribe();
  Output::ready(wait);

  theRFI.registerCreateInstance(sailingBoatClass.classHandleGet(), createSailboat);
```

Listing 5.6: Subscribing to Objects

Listing 5.7 on the next page shows a simple example of the factory method used in the example. The object factory creation function has to return an object of type `HLAObjectInstance` and its arguments are the handle of the object to be created and the class handle of the object. In this simple example the factory function increments an internal counter and creates a new `SailboatInstance`, which it returns. Generally, these factory functions don't do much more than what is shown here. Another possible task would be requesting some attribute updates for this object.

```
HLAObjectInstance* createSailboat(ftk::HANDLE objHandle, ftk::HANDLE classHandle)
{
   nb_discovered_boats++;
   the_last_boat = new SailboatInstance(objHandle, classHandle);
   return the_last_boat;
}
```

Listing 5.7: Defining the Object Factory

Because the Federate described in `listenBoat.cpp` subscribes to all attributes of Sailboat, it will be notified (reflection) each time an attribute is updated by the Federate owner of the corresponding attribute. The Federate getting the reflection can choose to be explicitly triggered on an attribute reflection by implementing the `reflect` method of the derived `HLAObjectInstance`. However, the default FTK-FederateAmbassador will set the correct attribute value of the specific object when it receives a reflection notification. Therefore, the developer should implement a reflect method only if there is a need for the program to respond specifically to the value change. Otherwise, the Federate can simply get the attribute value at any time with the `Get` method of each attribute, which returns the latest update of the value. The listenBoat example uses this scheme to print all attribute values of the last discovered boat.

Listing 5.8 shows how the status printing is done inside the simulation loop. The programs prints the retrieved information using `Get` methods, then “ticks” and goes to sleep for one second. In this case, the Federate is completely passive (it does not affect the simulation state), so there is no need to call `processPendings`. However, to be notified of the changes in the simulation, the Federate has to let the Federate Ambassador do the appropriate calls. This is achieved by ticking the RFI (`tick` method). Each time the Federate calls `tick`, the RFI lets its registered Federate Ambassador process all the pending calls it gets from the simulation. In our case, the Federate Ambassador reflects the attribute changes of the Sailboat objects.

```
      while ( theRFI.RFI::getNumberOfObjects() > 0 ) {
         cout << "Boat␣" << the_last_boat->nameGet()
              << "␣state:␣heading=" << the_last_boat->headingGet()
              << "␣/␣speed=" << the_last_boat->speedGet()
              << "␣/␣heel=" << the_last_boat->heelGet() << endl;
         RFI::tick();
         msSleep(1000);
      }
```

Listing 5.8: Listening to Attributes Updates

Note that the simulation loop of `createBoat` (Listing 5.5 on the preceding page did not tick at all because this Federate is only creating objects and generating updates. However, this practice is strongly discouraged for more complex simulations. Without ticking, the Federate never gives its Federate Ambassador a chance to process simulation requests and the Federates do not interact correctly in the simulation. In this particular case, `createBoat` would not answer to explicit requests for attribute updates. This is not critical here since the Federate already continuously updates all the attributes. However, imagine a situation where a first Federate updates attributes only when a change in its internal state occurs. A second Federate joining the simulation at a later time may depend on some of object’s attributes, but will not be able to get them until the first Federate decides to update them (which could be a long time if this particular attribute state does not change often). If the first Federate had been ticked, it could have updated the attribute that the second Federate requested and the simulation would have executed correctly.

5.5. Using O-O method calls between Federates

The examples shown in the previous sections demonstrate how to use the classic accessor methods for object attributes (valueSet – valueGet), but in this case between Federates that are connected via a communication network. This scheme is implemented on top of the HLA Object concept. FTK goes one step further by providing general O-O method calls (across the network) on instances that reside on a different Federate. This distributed calling feature is based on the HLA interaction concept, but additional mechanisms are embedded in FTK to turn HLA's "anonymous" interactions into MSF object method calls. The two following sections will explain how to use this capability through two additional example programs, `modelBoat.cpp` and `controlBoat.cpp`. The program listed in `modelBoat.cpp` creates a collection of Sailboat objects, and then simply waits for commands sent to these objects. When it receives a boat command, it prints the command to show the program's behavior. On the other side, the `controlBoat` program discovers the boats that are created in the simulation and sends commands to these boats with a method call. After each method call, the program waits to receive an acknowledgment that the command was received by the program that models the boats.

5.5.1. Calling Remote Object Methods

Calling a method on a remote object is straightforward as shown in Listing 5.9. If the program has a reference on a *discovered*[6] object instance, it can simply call the method attached to this object with the necessary arguments. In response to the method call, the object instance builds the appropriate message (HLA interaction) and sends it to the simulation. The message carries the handle of the object as well as a command identifier. The object handle is used by the receiving Federate to route the message to the designated object. The command identifier is uniquely defined (among all Federates) by the sending Federate, and serves two purposes: 1) the receiving Federate needs this command identifier to create a `ReturnValue` object instance, and 2) the sending Federate can query its RFI for the status of this particular call. A remote method call on an `HLAObjectInstance` is not blocking, so the method returns as soon as the message is sent. In addition, all `HLAObjectInstance` remote method calls return the command identifier, which has been automatically generated so that it can be used to query the command status.

```
    for (int i=0; i<nb_boats; i++) {
        sprintf(name, "boat_%d", i);
        HLAObjectInstance* obj = theRFI.getObjectInstance(name);
        SailboatInstance* boat = dynamic_cast<SailboatInstance*>(obj);
        if ( boat ) {

            sprintf(anchor, "A_%c", char(i+65));
            cout << "Drop␣Anchor␣" << anchor << "␣with␣"
                 << (i+1)*10 << "␣meter␣of␣chain␣" << "␣of␣boat␣" << name << endl;
            cmdID = boat->dropAnchor(string(anchor), (float)(i+1)*10);
            control_wait_for_acknowledgment(cmdID);
```

Listing 5.9: Calling Object Methods

It should be noted that the remote method scheme relies on HLA interactions, which implies that the Federate needs to declare its intention to publish the interaction supporting the remote method call. In our

[6]This is currently not enforced by FTK itself: it is possible to call remote method on object created by the Federate itself. We should change this behavior by inserting a test in the code generation process.

example `SailboatSetSailInteraction` needs to be declared if the `setSail` method is used (see Listing B.4 on page 45 for details on this).

The sending Federate can check the status of the remote method call by querying the RFI with `getReturnValue`. Before the `ReturnValue` instance has been created by the receiving Federate, `getReturnValue` will return `-1`. Once the `ReturnValue` has been created by the receiving Federate, and discovered by the sending Federate, `getReturnValue` will return the value of the attribute, `state` of the `ReturnValue` instance. Sample code showing how to wait for return values is shown in Listing 5.10. The example includes a time-out loop, since there is no guarantee that the receiving Federate will create the appropriate `ReturnValue`. Again, it is necessary to "tick" the RFI inside this waiting loop, since it is the only way for the RTI to notify the RFI of changes in the simulation state.

```
bool control_wait_for_acknowledgment(msf::U_LONG cmdID)
{
      bool acknowledge = false;
      long sec, msec;
      Timer timer;

      for (int t=0; t<1000 && !acknowledge; t++) {
         RFI::tick();
         if ( -1 != RFI::getReturnValue(cmdID) ) {
            acknowledge = true;
         }
         msSleep(5);
      }
      if ( acknowledge ) {
         timer.elapsed(sec, msec);
         cout << "␣␣->␣acknowledgment␣in␣"
              << sec << "s␣and␣" << msec << "ms" << endl;
         return true;
      }
      else {
         cout << "Time␣out␣for␣acknowledgment!" << endl;
         return false;
```

Listing 5.10: Waiting for Acknowledgment

5.5.2. Implementing Object Methods

On the receiving side (i.e. the Federate modeling the actual instance), some additional work is required to handle the remote method calls. In addition to declaring the Federate's intention to subscribe to the necessary interactions (see Listing B.3 on page 43 for details), the Federate has to initialize the interaction factory. This scheme works exactly the same way as the object factory explained in Section 5.4 on page 22: the Federate registers to the RFI how to create interaction messages of various classes using the `registerCreateMessage` method. An example of a message factory function is shown in Listing 5.11. When receiving a message of class `SailboatSetSail`, the function simply creates a `SailboatSetSailMessage`, which is added automatically to the list of pending messages in the RFI.

Of course, the receiving Federate also has to implement the setSail method requested by the sending Federate. For that purpose, one has to derive a class from the base instance class and implement the required remote method calls. The code in Listing 5.12 on the next page illustrates one of these implementations. The function simply creates the appropriate return value and prints out a small message showing the arguments

```
HLAInteractionMessage* createSetSail(ftk::HANDLE classHandle)
{
   return new SailboatSetSailMessage(classHandle);
}
```

Listing 5.11: Message Factory

received.

```
class MyBoat : public SailboatInstance
{
   public:
      MyBoat(const char* name) {
         nameSet(name);
      };

      void setSail(SailboatSetSailMessage* msg) {
         RFI::createReturnValue(msg->commandIdentifierGet());
         cout << "MyBoat␣" << nameGet()
              << "␣is␣commanded␣to␣set␣sail␣" << msg->sailGet() << "␣!" << endl;
      }
```

Listing 5.12: Implementation of the Methods

Because the `MyBoat` class implements the remote methods declared in `SailboatInstance` to achieve the expected behavior, the Federate has to create objects of type `MyBoat` rather than the generic `SailboatInstance` as described in Listing 5.13. It should be understood that although this Federate registers object instances of type `MyBoat` to the RFI, in actuality instances of class `SailboatInstance` are broadcast to the simulation and the `MyBoat` class implementation exists only locally to this specific Federate.

```
   for (int i=0; i<nb_boats; i++) {
      char name[32];
      sprintf(name, "boat_%d", i);
      MyBoat* boat = new MyBoat(name);
      theRFI.registerObjectInstance(boat);
   }
```

Listing 5.13: Creating Object Instance Implementing the Remote Method Calls

Finally, the receiving/modeling Federate can enter into its simulation loop as depicted in Listing 5.14. In this simulation loop, the Federate simply executes the `tick` and `processPendings` commands, which ensure that:

- The Federate receives remote method calls (through interaction message)
- The RFI processes the local pending actions, like registering to the RTI a newly created `ReturnValue`

```
   for (int t=0; t<3000; t++) {
      RFI::processPendings();
      RFI::tick();
      // sleep for 100 milliseconds
      msSleep(100);
```

```
}
```

Listing 5.14: Simulation Loop of the Model

5.6. Object Shells

The publish/subscribe scheme for objects is a satisfactory solution in all situations where a Federate needs to be notified about all the objects of a given class. For example, a viewer would subscribe to all graphical objects and display them indiscriminately of their particular identity. There are however many situations where a component needs to get or set the data of a specific object. For example, a trajectory generator that computes the wheel movement of a rover, when it receives a command "Drive to this position", needs to command the individual wheels of a particular rover, not just any wheel belonging to a rover. Because the models of the rover wheels are computed by another component (e.g. the dynamic simulator), they are not created by the trajectory generator. Therefore, the trajectory generator needs to discover the correct wheels it wants to control. A Federate could perform this task by trying to identify each newly discovered object and comparing it to the Federate's internal list of objects that it wants to control. FTK provides a facility to implement this in a more generic way with a concept termed Object Shells, or simply Shell.

A Shell is an object that has a reference to a discovered object. A Shell is initially empty and contains only the name of the object it is waiting for. By registering the Shell to the RFI, it will start monitoring any new discovery of objects having the same name as the Shell. When the RFI is notified of a new object in the simulation, it automatically tries to match it to the list of registered Shells. If a name matches, it fills the Shell with the reference to the discovered object, making this Shell usable by the component. If the object is removed from the simulation, the Shell is notified and emptied. but will continue to be active and will be filled again on object re-discovery or emptied on object removal, as long as it stays registered to the RFI. To make the Shell concept even more useful, a `Discovery Policy` is attached to every `Shell` instructing the RFI to perform additional actions upon Shell discovery, including:

- Requesting updates for a set of attributes (necessary if the component needs some attribute value of the object right at the initialization phase)
- Requesting ownership for a set of attributes (necessary if the component wants to take control of these attributes)

The Shell becomes "ready" only when all the following conditions are met:

- The object instance with the correct name has been discovered
- The attribute update requests have been satisfied
- The attribute ownership requests have been satisfied

The Shell concept, which drastically decreases the amount of code required by the simulation component developer, is implemented with the help of the templatized class `TObjShell` and the policy class `DiscoverPolicy`. Listing 5.15 on the following page illustrates how to use them. The fully documented listing of this program, `discoverSushis`, which is to be used together with `createSushis`, is given in Listing B.5 on page 48. The first lines of the example simply declare two different policies. The makiPolicy instanct requires an attribute update for `PIECES` and the nigiriPolicy instant requires attribute ownership of

PAIR. Two Shells are then built with these policies. The first Shell will wait for an object named “kapa” and the second Shell will wait for an object identified with the name “maguro”. Finally, the two Shells are registered to the RFI in the last lines of the example.

```
DiscoverPolicy makiPolicy;
makiPolicy.addUpdateRequest(makiClass.getAttributeHandle(msf::MakiObjClass::PIECES))

DiscoverPolicy nigiriPolicy;
nigiriPolicy.addOwnershipRequest(
    nigiriClass.getAttributeHandle(msf::NigiriObjClass::PAIR));
TObjShell<msf::MakiInstance> makiShell("kapa", makiPolicy);
TObjShell<msf::NigiriInstance> nigiriShell("maguro", nigiriPolicy);

rfi.registerObjShell(&makiShell);
rfi.registerObjShell(&nigiriShell);
```

Listing 5.15: Creating and Registering Shell

Bibliography

[1] Defense Modeling and Simulation Office. *RTI 1.3-Next Generation Programmer's Guide*. URL=http://www.dmso.mil/public/transition/hla/.

[2] Lorenzo Flückiger. *Generating HLA based communication classes form a UML description*. NASA Ames Research Center, 2002. URL=http://is.arc.nasa.gov/MSF...

[3] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating Computer Simulations Systems: An Introduction to the High Level Architecture*. Prentice Hall, 2000.

A. UML Sequences Diagrams for FTK

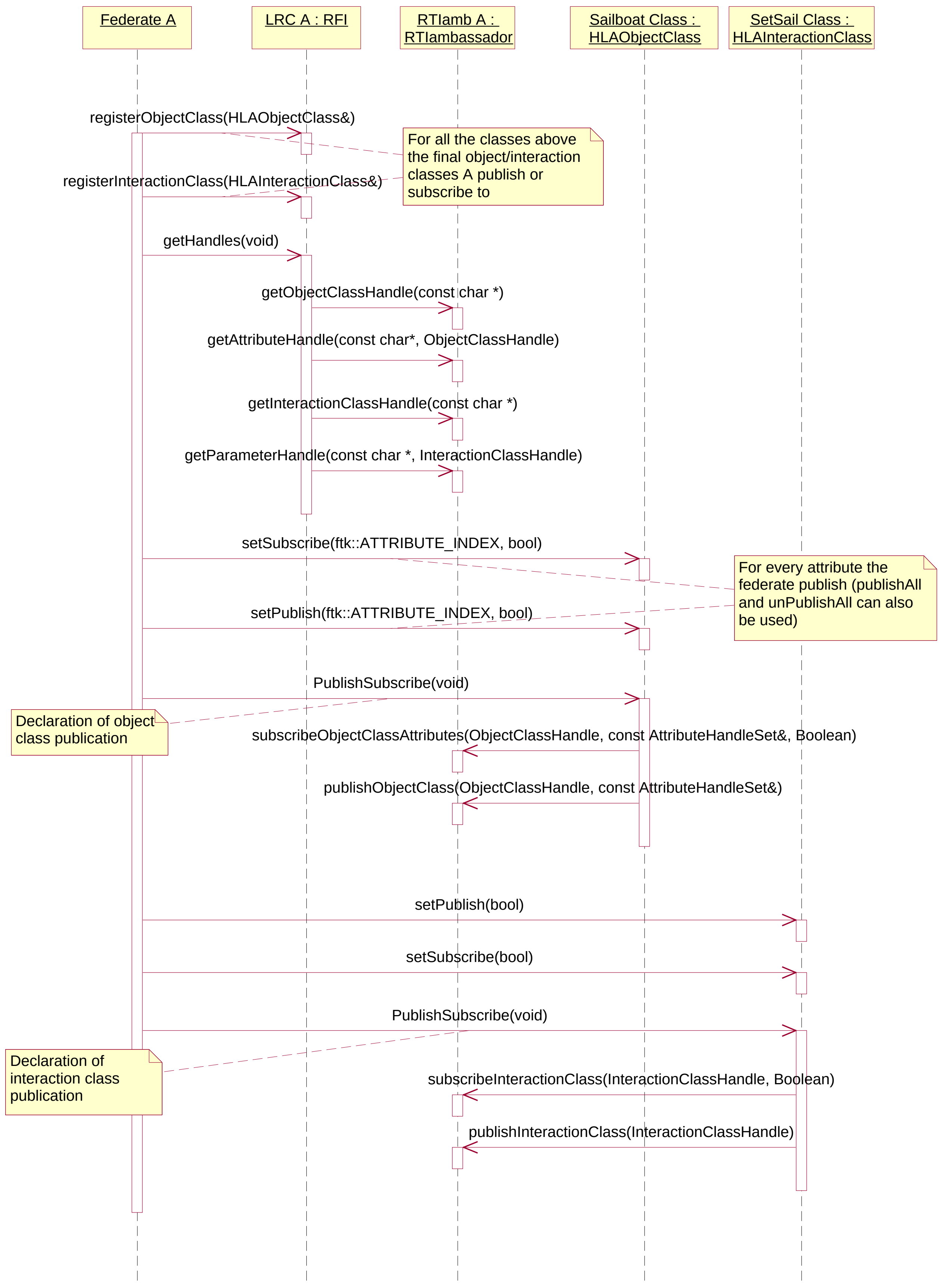

Federate A
LRC A : RFI
RTIamb A : RTIambassador
Sailboat Class : HLAObjectClass
SetSail Class : HLAInteractionClass
registerObjectClass(HLAObjectClass&)
For all the classes above the final object/interaction classes A publish or subscribe to
registerInteractionClass(HLAInteractionClass&)
getHandles(void)
getObjectClassHandle(const char *)
getAttributeHandle(const char*, ObjectClassHandle)
getInteractionClassHandle(const char *)
getParameterHandle(const char *, InteractionClassHandle)
setSubscribe(ftk::ATTRIBUTE_INDEX, bool)
For every attribute the federate publish (publishAll and unPublishAll can also be used)
setPublish(ftk::ATTRIBUTE_INDEX, bool)
PublishSubscribe(void)
Declaration of object class publication
subscribeObjectClassAttributes(ObjectClassHandle, const AttributeHandleSet&, Boolean)
publishObjectClass(ObjectClassHandle, const AttributeHandleSet&)
setPublish(bool)
setSubscribe(bool)
PublishSubscribe(void)
Declaration of interaction class publication
subscribeInteractionClass(InteractionClassHandle, Boolean)
publishInteractionClass(InteractionClassHandle)

Figure A.1.: Declaration Management.

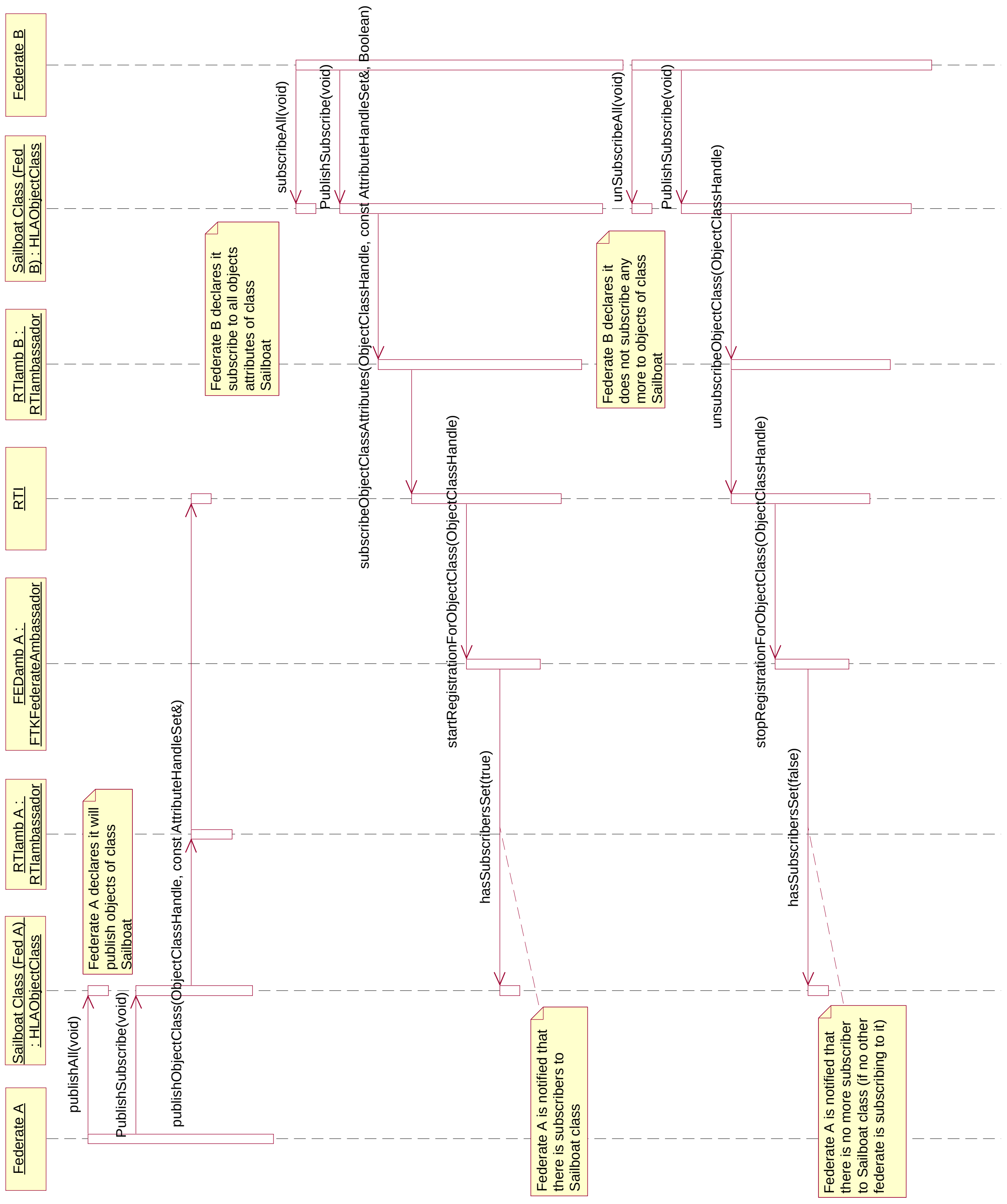

Federate B
Sailboat Class (Fed B) : HLAObjectClass
RTIamb B : RTIambassador
RTI
FEDamb A : FTKFederateAmbassador
RTIamb A : RTIambassador
Sailboat Class (Fed A) : HLAObjectClass
Federate A
subscribeAll(void)
PublishSubscribe(void)
subscribeObjectClassAttributes(ObjectClassHandle, const AttributeHandleSet&, Boolean)
startRegistrationForObjectClass(ObjectClassHandle)
hasSubscribersSet(true)
unSubscribeAll(void)
PublishSubscribe(void)
unsubscribeObjectClass(ObjectClassHandle)
stopRegistrationForObjectClass(ObjectClassHandle)
hasSubscribersSet(false)
publishAll(void)
PublishSubscribe(void)
publishObjectClass(ObjectClassHandle, const AttributeHandleSet&)
Federate B declares it subscribe to all objects attributes of class Sailboat
Federate B declares it does not subscribe any more to objects of class Sailboat
Federate A declares it will publish objects of class Sailboat
Federate A is notified that there is subscribers to Sailboat class
Federate A is notified that there is no more subscriber to Sailboat class (if no other federate is subscribing to it)

Figure A.2.: Declaration Propagation and Notification.

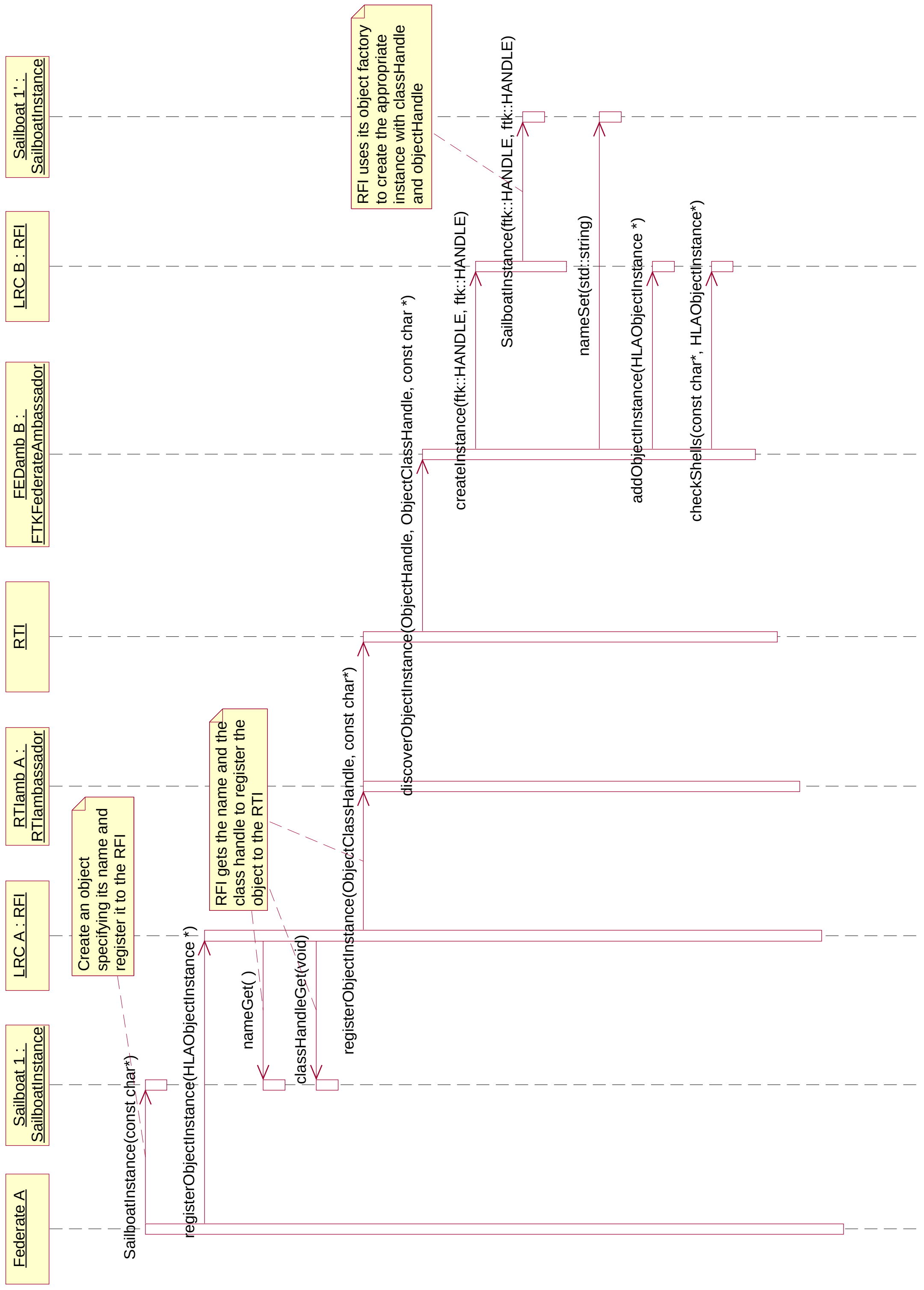

Federate A
Sailboat 1 : SailboatInstance
LRC A : RFI
RTIamb A : RTIambassador
RTI
FEDamb B : FTKFederateAmbassador
LRC B : RFI
Sailboat 1' : SailboatInstance
Create an object specifying its name and register it to the RFI
RFI gets the name and the class handle to register the object to the RTI
RFI uses its object factory to create the appropriate instance with classHandle and objectHandle
SailboatInstance(const char*)
registerObjectInstance(HLAObjectInstance *)
nameGet( )
classHandleGet(void)
registerObjectInstance(ObjectClassHandle, const char*)
discoverObjectInstance(ObjectHandle, ObjectClassHandle, const char *)
createInstance(ftk::HANDLE, ftk::HANDLE)
SailboatInstance(ftk::HANDLE, ftk::HANDLE)
nameSet(std::string)
addObjectInstance(HLAObjectInstance *)
checkShells(const char*, HLAObjectInstance*)

Figure A.3.: Object Creation.

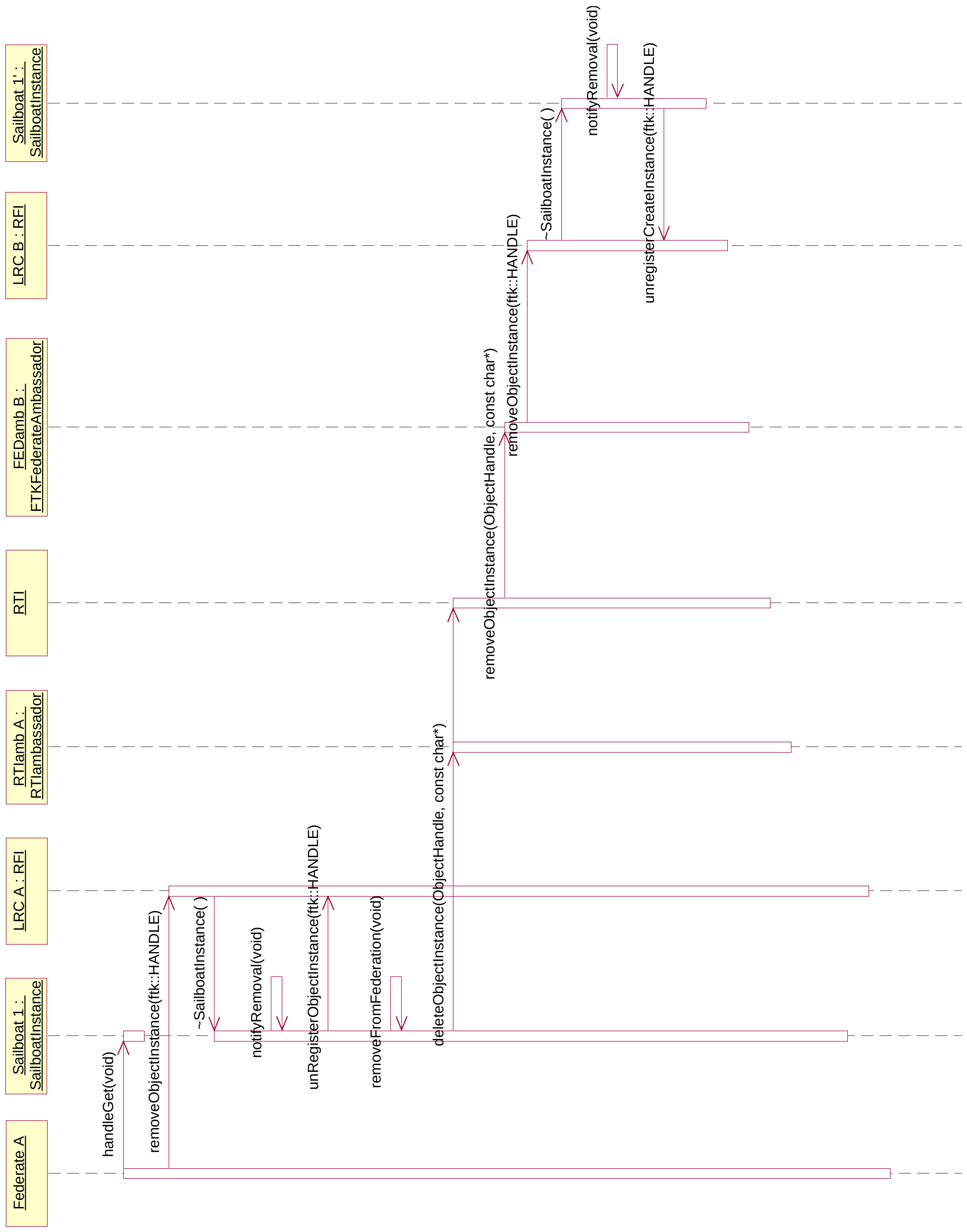

Federate A
Sailboat 1 : SailboatInstance
LRC A : RFI
RTIamb A : RTIambassador
RTI
FEDamb B : FTKFederateAmbassador
LRC B : RFI
Sailboat 1' : SailboatInstance
handleGet(void)
removeObjectInstance(ftk::HANDLE)
~SailboatInstance( )
notifyRemoval(void)
unRegisterObjectInstance(ftk::HANDLE)
removeFromFederation(void)
deleteObjectInstance(ObjectHandle, const char*)
removeObjectInstance(ObjectHandle, const char*)
removeObjectInstance(ftk::HANDLE)
~SailboatInstance( )
notifyRemoval(void)
unregisterCreateInstance(ftk::HANDLE)

Figure A.4.: Object Deletion.

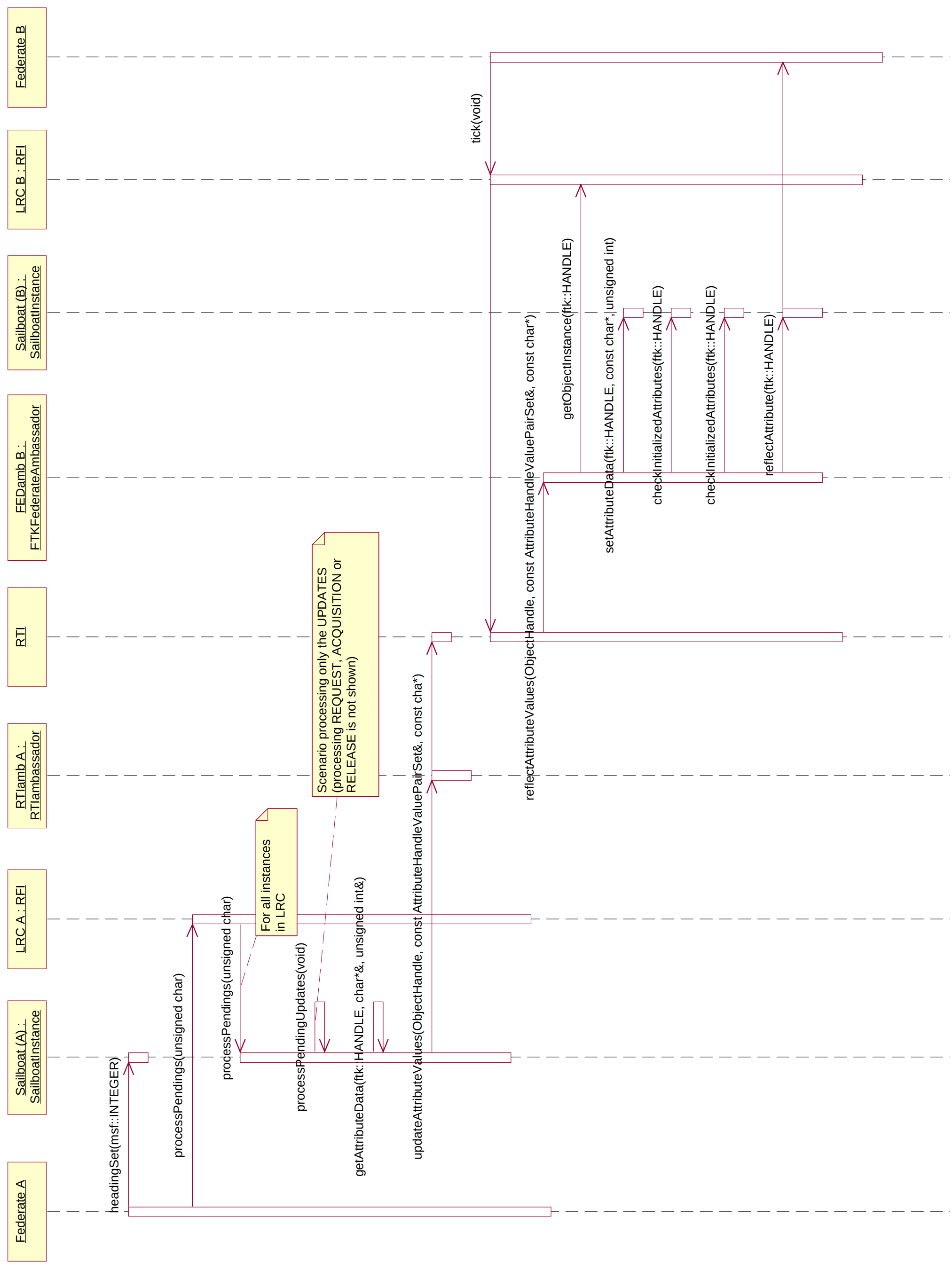
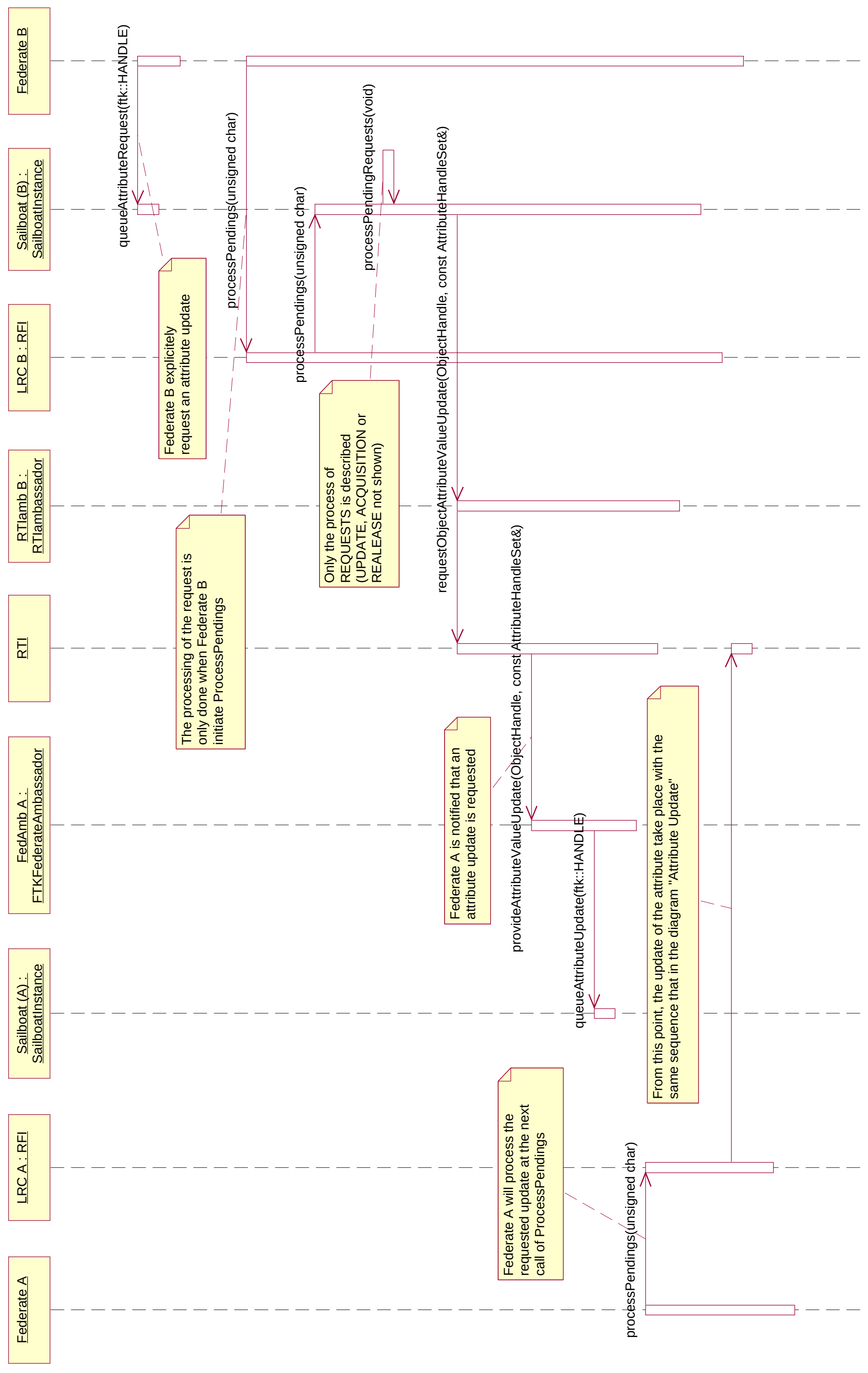

Federate A
Sailboat (A) : SailboatInstance
LRC A : RFI
RTIamb A : RTIambassador
RTI
FEDamb B : FTKFederateAmbassador
Sailboat (B) : SailboatInstance
LRC B : RFI
Federate B
headingSet(msf::INTEGER)
processPendings(unsigned char)
processPendings(unsigned char)
For all instances in LRC
processPendingUpdates(void)
Scenario processing only the UPDATES (processing REQUEST, ACQUISITION or RELEASE is not shown)
getAttributeData(ftk::HANDLE, char*&, unsigned int&)
updateAttributeValues(ObjectHandle, const AttributeHandleValuePairSet&, const char*)
tick(void)
reflectAttributeValues(ObjectHandle, const AttributeHandleValuePairSet&, const char*)
getObjectInstance(ftk::HANDLE)
setAttributeData(ftk::HANDLE, const char*, unsigned int)
checkInitializedAttributes(ftk::HANDLE)
checkInitializedAttributes(ftk::HANDLE)
reflectAttribute(ftk::HANDLE)

Figure A.5.: Attribute Update.



Federate B
Sailboat (B) : SailboatInstance
LRC B : RFI
RTIamb B : RTIambassador
RTI
FedAmb A : FTKFederateAmbassador
Sailboat (A) : SailboatInstance
LRC A : RFI
Federate A
queueAttributeRequest(ftk::HANDLE)
Federate B explicitely request an attribute update
processPendings(unsigned char)
The processing of the request is only done when Federate B initiate ProcessPendings
processPendings(unsigned char)
processPendingRequests(void)
Only the process of REQUESTS is described (UPDATE, ACQUISITION or REALEASE not shown)
requestObjectAttributeValueUpdate(ObjectHandle, const AttributeHandleSet&)
provideAttributeValueUpdate(ObjectHandle, const AttributeHandleSet&)
Federate A is notified that an attribute update is requested
queueAttributeUpdate(ftk::HANDLE)
From this point, the update of the attribute take place with the same sequence that in the diagram "Attribute Update"
Federate A will process the requested update at the next call of ProcessPendings
processPendings(unsigned char)

Figure A.6.: Attribute Request.

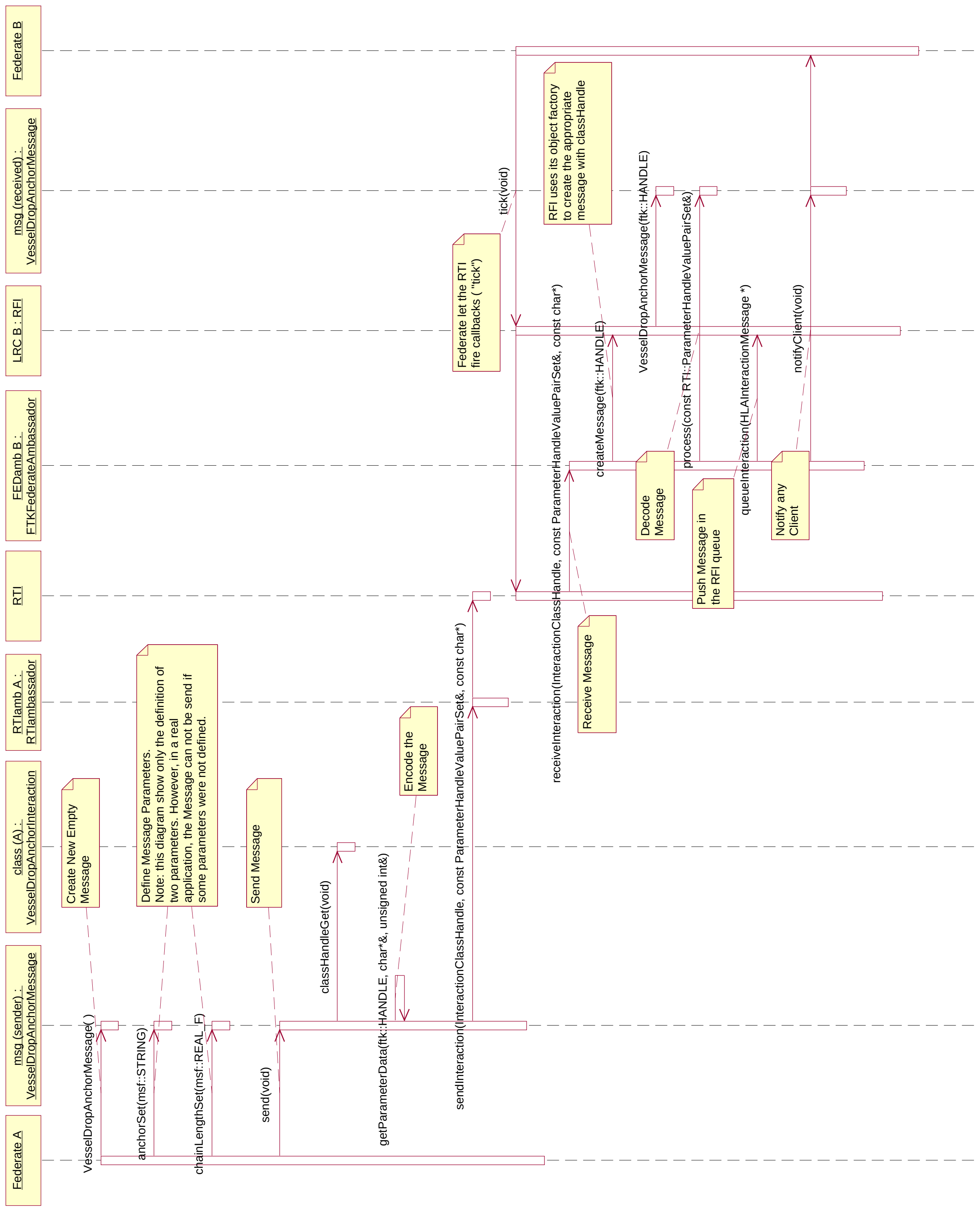
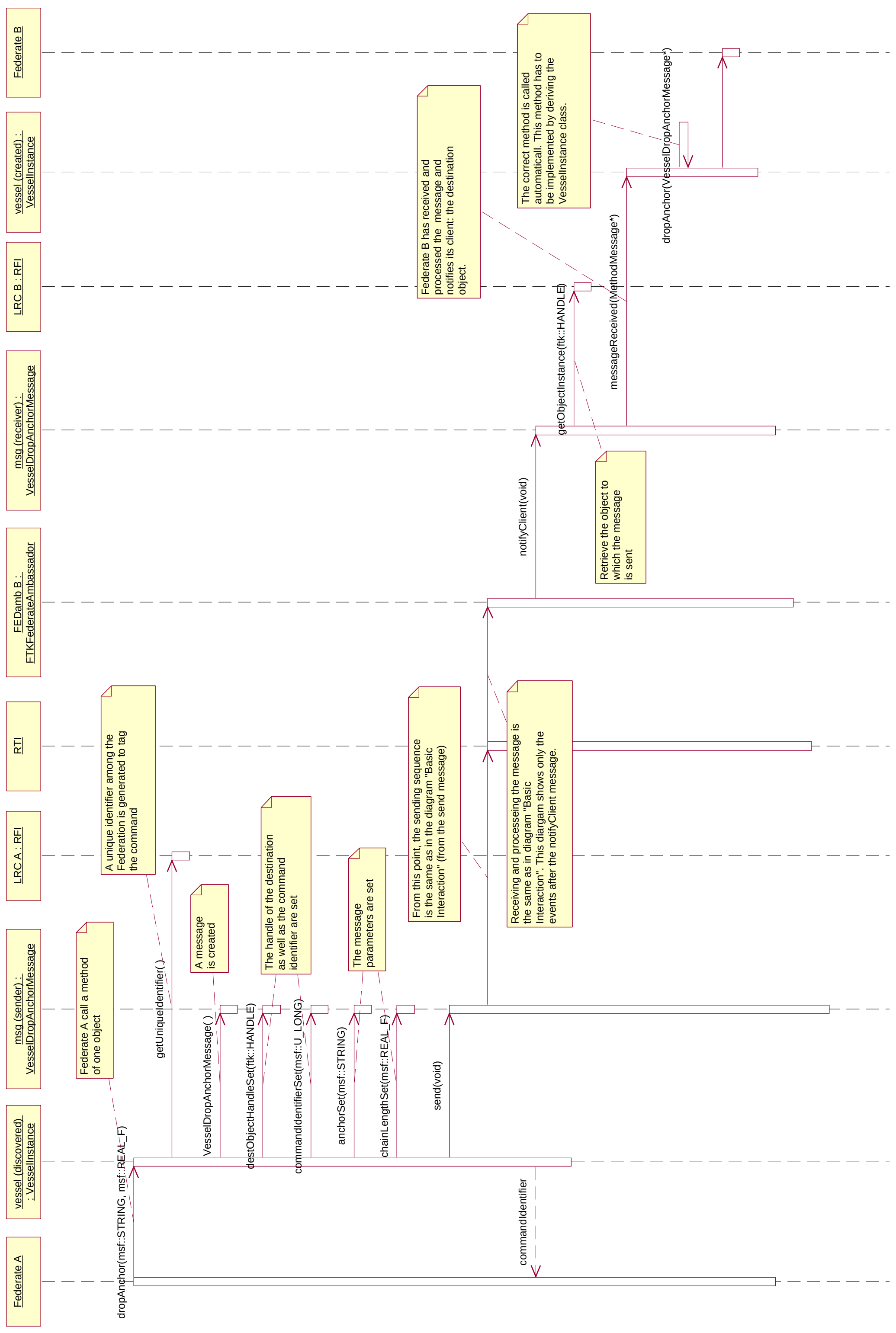

Federate A
msg (sender) : VesselDropAnchorMessage
class (A) : VesselDropAnchorInteraction
RTIamb A : RTIambassador
RTI
FEDamb B : FTKFederateAmbassador
LRC B : RFI
msg (received) : VesselDropAnchorMessage
Federate B
Create New Empty Message
Define Message Parameters. Note: this diagram show only the definition of two parameters. However, in a real application, the Message can not be send if some parameters were not defined.
Send Message
Encode the Message
Receive Message
Federate let the RTI fire callbacks ("tick")
RFI uses its object factory to create the appropriate message with classHandle
Decode Message
Push Message in the RFI queue
Notify any Client
VesselDropAnchorMessage( )
anchorSet(msf::STRING)
chainLengthSet(msf::REAL_F)
send(void)
classHandleGet(void)
getParameterData(ftk::HANDLE, char*&, unsigned int&)
sendInteraction(InteractionClassHandle, const ParameterHandleValuePairSet&, const char*)
tick(void)
receiveInteraction(InteractionClassHandle, const ParameterHandleValuePairSet&, const char*)
createMessage(ftk::HANDLE)
VesselDropAnchorMessage(ftk::HANDLE)
process(const RTI::ParameterHandleValuePairSet&)
queueInteraction(HLAInteractionMessage *)
notifyClient(void)

Figure A.7.: Basic Interaction.


Federate A
vessel (discovered) : VesselInstance
msg (sender) : VesselDropAnchorMessage
LRC A : RFI
RTI
FEDamb B : FTKFederateAmbassador
msg (receiver) : VesselDropAnchorMessage
LRC B : RFI
vessel (created) : VesselInstance
Federate B
dropAnchor(msf::STRING, msf::REAL_F)
getUniqueIdentifier( )
VesselDropAnchorMessage( )
destObjectHandleSet(ftk::HANDLE)
commandIdentifierSet(msf::U_LONG)
anchorSet(msf::STRING)
chainLengthSet(msf::REAL_F)
send(void)
commandIdentifier
notifyClient(void)
getObjectInstance(ftk::HANDLE)
messageReceived(MethodMessage*)
dropAnchor(VesselDropAnchorMessage*)

Federate A call a method of one object

A unique identifier among the Federation is generated to tag the command

A message is created

The handle of the destination as well as the command identifier are set

The message parameters are set

From this point, the sending sequence is the same as in the diagram "Basic Interaction" (from the send message)

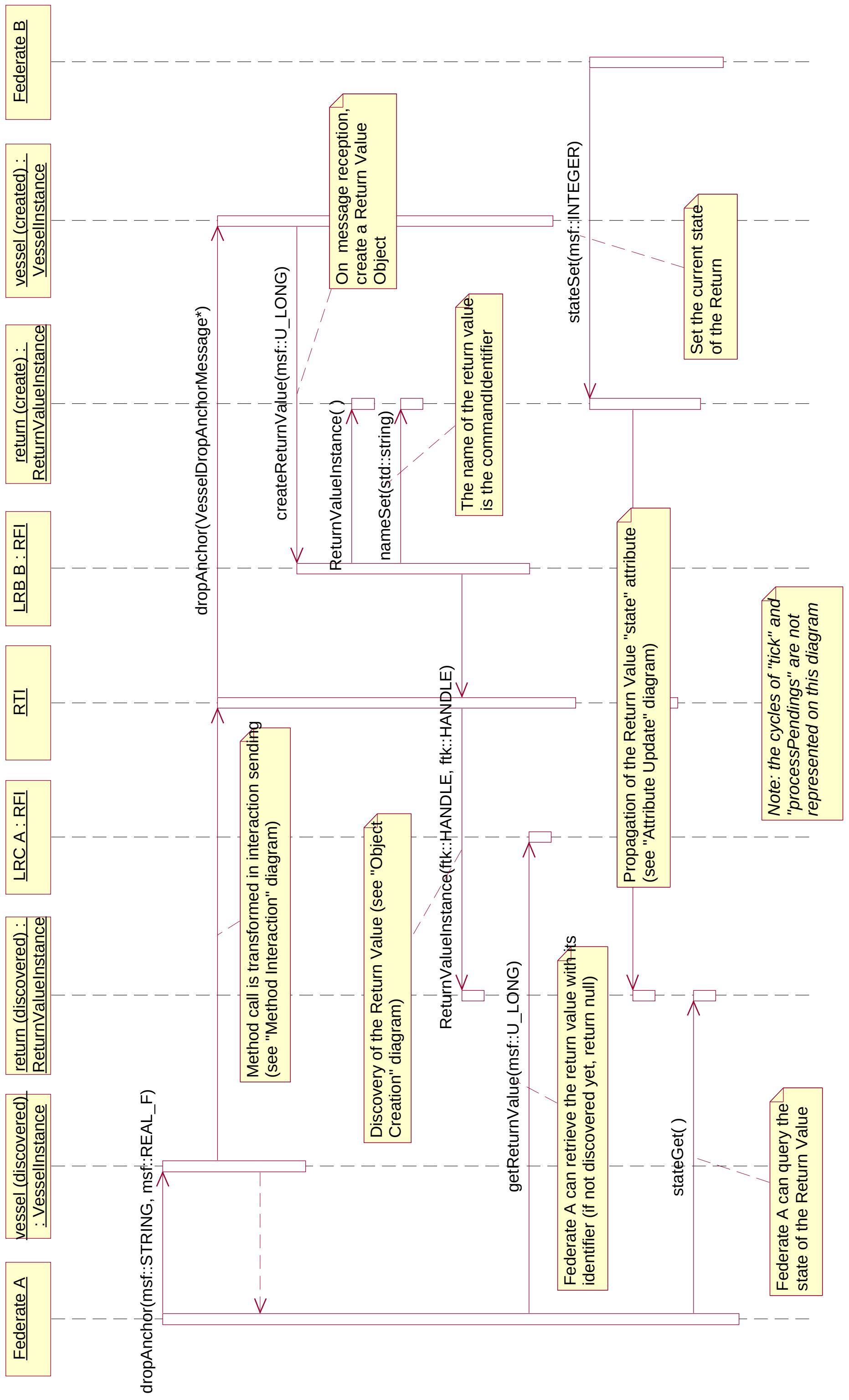
Receiving and processeing the message is the same as in diagram "Basic Interaction". This diagaram shows only the events after the notifyClient message.

Retrieve the object to which the message is sent

Federate B has received and processed the message and notifies its client: the destination object.

The correct method is called automaticall. This method has to be implemented by deriving the VesselInstance class.

Figure A.8.: Method Interaction.



Federate A
vessel (discovered) : VesselInstance
return (discovered) : ReturnValueInstance
LRC A : RFI
RTI
LRB B : RFI
return (create) : ReturnValueInstance
vessel (created) : VesselInstance
Federate B
dropAnchor(msf::STRING, msf::REAL_F)
dropAnchor(VesselDropAnchorMessage*)
Method call is transformed in interaction sending (see "Method Interaction" diagram)
createReturnValue(msf::U_LONG)
On message reception, create a Return Value Object
ReturnValueInstance( )
nameSet(std::string)
The name of the return value is the commandIdentifier
Discovery of the Return Value (see "Object Creation" diagram)
ReturnValueInstance(ftk::HANDLE, ftk::HANDLE)
getReturnValue(msf::U_LONG)
Federate A can retrieve the return value with its identifier (if not discovered yet, return null)
stateSet(msf::INTEGER)
Set the current state of the Return
Propagation of the Return Value "state" attribute (see "Attribute Update" diagram)
stateGet( )
Federate A can query the state of the Return Value
Note: the cycles of "tick" and "processPendings" are not represented on this diagram

Figure A.9.: Return Value.

B. Code of Examples

Listing B.1: Create and Update — createBoat.cpp

```
// !!! WARNING !!!
// Please never add or remove any line from this file without updating the
// msf.tex documentation accordingly (Lorenzo)

#include "rfi.h"
#include "ftkfederateambassador.h"

#include "VesselObjClass.h"
#include "SailboatObjClass.h"
#include "SailboatInstance.h"

using namespace msf;

int main (unsigned int argc, char *argv[])
{
   RFI theRFI;
   FTKFederateAmbassador* theFA = new FTKFederateAmbassador();
   theRFI.federateAmbassadorSet(theFA);

   theRFI.federationNameSet("MSF-TEST");
   theRFI.federationFileSet("navigation.fed");

   HLAObjectClass& vesselClass = SingletonHolder<VesselObjClass>::Instance();
   HLAObjectClass& sailingBoatClass = SingletonHolder<SailboatObjClass>::Instance();
   theRFI.registerObjectClass(vesselClass);
   theRFI.registerObjectClass(sailingBoatClass);

   int wait = Output::wait("Trying␣to␣Join␣the␣Federation");
   int error =  theRFI.joinFederation();
   if ( error ) {
      Output::fatal("An␣Error␣has␣occured␣when␣joining:␣stop␣the␣program!\n");
   }
   else {
      Output::ready(wait);
   }

   wait = Output::wait("Getting␣all␣handles");
   theRFI.getHandles();
   Output::ready(wait);

   wait = Output::wait("Publishing␣federates␣classes");
   sailingBoatClass.publishAll();
   sailingBoatClass.unSubscribeAll();
   sailingBoatClass.PublishSubscribe();
   Output::ready(wait);
```

```
    wait = Output::wait("Wait␣for␣subscribers␣of␣Sailboat");
    for (int t=0; t<100 && !sailingBoatClass.hasSubscribersGet(); t++) {
        RFI::tick();
        RFI::processPendings();
        msSleep(1000);
    }

    if ( sailingBoatClass.hasSubscribersGet() ) {
        Output::ready(wait);

        wait = Output::wait("Creating␣one␣Sailboat");
        SailboatInstance* boat = new SailboatInstance();
        boat->nameSet("my_best_boat");
        theRFI.registerObjectInstance(boat);
        Output::ready(wait);

        wait = Output::wait("Updating␣boat␣attributes");
        for (int n=0; n<100; n++) {
            boat->headingSet(270);
            boat->speedSet(n%20);
            boat->heelSet((int)((n-50)/10));
            RFI::processPendings();
            msSleep(1000);
        }
        Output::ready(wait);
    }
    else {
        Output::error("Time␣Out!\n");
        return -1;
    }

    return 0;

}
```

Listing B.2: Discover and Reflect — listenBoat.cpp

```
// !!! WARNING !!!
// Please never add or remove any line from this file without updating the
// msf.tex documentation accordingly (Lorenzo)

#include "rfi.h"
#include "ftkfederateambassador.h"

#include "VesselObjClass.h"
#include "SailboatObjClass.h"
#include "SailboatInstance.h"

using namespace msf;

int nb_discovered_boats = 0;
SailboatInstance* the_last_boat = 0;

HLAObjectInstance* createSailboat(ftk::HANDLE objHandle, ftk::HANDLE classHandle)
{
    nb_discovered_boats++;
    the_last_boat = new SailboatInstance(objHandle, classHandle);
    return the_last_boat;
}

int main (unsigned int argc, char *argv[])
{
    RFI theRFI;
    FTKFederateAmbassador* theFA = new FTKFederateAmbassador();
    theRFI.federateAmbassadorSet(theFA);

    theRFI.federationNameSet("MSF-TEST");
    theRFI.federationFileSet("navigation.fed");

    HLAObjectClass& vesselClass = SingletonHolder<VesselObjClass>::Instance();
    HLAObjectClass& sailingBoatClass = SingletonHolder<SailboatObjClass>::Instance();
    theRFI.registerObjectClass(vesselClass);
    theRFI.registerObjectClass(sailingBoatClass);

    int wait = Output::wait("Trying␣to␣Join␣the␣Federation");
    int error =  theRFI.joinFederation();
    if ( error ) {
        Output::fatal("An␣Error␣as␣occured␣when␣joining:␣stop␣the␣program!\n");
    }
    else {
        Output::ready(wait);
    }

    wait = Output::wait("Getting␣all␣handles");
    theRFI.getHandles();
    Output::ready(wait);

    wait = Output::wait("Publishing␣federates␣classes");
    sailingBoatClass.unPublishAll();
    sailingBoatClass.subscribeAll();
    sailingBoatClass.PublishSubscribe();
    Output::ready(wait);

    theRFI.registerCreateInstance(sailingBoatClass.classHandleGet(), createSailboat);

    wait = Output::wait("Wait␣for␣one␣boat");
    for (int t=0; t<100 && nb_discovered_boats<1; t++) {
        RFI::tick();
        msSleep(1000);
    }

    if ( nb_discovered_boats > 0 ) {
        Output::ready(wait);

        while ( theRFI.RFI::getNumberOfObjects() > 0 ) {
            cout << "Boat␣" << the_last_boat->nameGet()
                 << "␣state:␣heading=" << the_last_boat->headingGet()
                 << "␣/␣speed=" << the_last_boat->speedGet()
                 << "␣/␣heel=" << the_last_boat->heelGet() << endl;
            RFI::tick();
            msSleep(1000);
        }

    }
    else {
        Output::error("Time Out!\n");
        return -1;
    }

    return 0;

}
```

Listing B.3: Model a Boat Responding to Method Calls — modelBoat.cpp

```
// !!! WARNING !!!
// Please never add or remove any line from this file without updating the
// msf.tex documentation accordingly (Lorenzo)

#include "rfi.h"
#include "ftkfederateambassador.h"

#include "VesselDropAnchorMessage.h"
#include "VesselWeighAnchorsMessage.h"
#include "SailboatSetSailMessage.h"
#include "SailboatInstance.h"

#include "classes_hierarchy.h"

#include <stdio.h>  // sprintf...

HLAInteractionMessage* createSetSail(ftk::HANDLE classHandle)
{
    return new SailboatSetSailMessage(classHandle);
}

HLAInteractionMessage* createDropAnchor(ftk::HANDLE classHandle)
{
    return new VesselDropAnchorMessage(classHandle);
}

HLAInteractionMessage* createWeighAnchors(ftk::HANDLE classHandle)
{
    return new VesselWeighAnchorsMessage(classHandle);
}


class MyBoat : public SailboatInstance
{
    public:
        MyBoat(const char* name) {
            nameSet(name);
        };

        void setSail(SailboatSetSailMessage* msg) {
            RFI::createReturnValue(msg->commandIdentifierGet());
            cout << "MyBoat␣" << nameGet()
                 << "␣is␣commanded␣to␣set␣sail␣" << msg->sailGet() << "␣!" << endl;
        }

        void weighAnchors(VesselWeighAnchorsMessage* msg) {
            RFI::createReturnValue(msg->commandIdentifierGet());
            cout << "MyBoat␣" << nameGet()
                 << "␣is␣commanded␣to␣weigh␣anchors␣!" << endl;
        }

        void dropAnchor(VesselDropAnchorMessage* msg) {
            RFI::createReturnValue(msg->commandIdentifierGet());
            cout << "MyBoat␣" << nameGet()
                 << "␣is␣commanded␣to␣drop␣anchor␣" << msg->anchorGet()
                 << "␣with␣" << msg->chainLengthGet() << "␣of␣chain!" << endl;
        }


};

int main (unsigned int argc, char *argv[])
{

    if ( argc < 2 ) {
        cerr << "Usage:␣" << argv[0] << "␣number_of_boats" << endl;
        exit(1);
    }
    int nb_boats = atoi(argv[1]);

    RFI theRFI;
    FTKFederateAmbassador* theFA = new FTKFederateAmbassador();
    theRFI.federateAmbassadorSet(theFA);

    theRFI.federationNameSet("MSF-TEST");
    theRFI.federationFileSet("navigation.fed");

    int wait = Output::wait("Trying␣to␣Join␣the␣Federation");
    int error =  theRFI.joinFederation();
    if ( error ) {
        Output::fatal("An␣Error␣as␣occured␣when␣joining:␣stop␣the␣program!\n");
    }
    else {
        Output::ready(wait);
    }

    wait = Output::wait("Getting␣all␣handles");
    buildClassesHierarchy();
    theRFI.getHandles();
    Output::ready(wait);

    wait = Output::wait("Publishing␣federates␣classes");
    sailingBoatClass->publishAll();
    sailingBoatClass->unSubscribeAll();
    sailingBoatClass->PublishSubscribe();
    dropAnchorInteraction->PublishSubscribe();
    weighAnchorsInteraction->PublishSubscribe();
    setSailInteraction->PublishSubscribe();

    HLAObjectClass& returnClass = SingletonHolder<ReturnValueObjClass>::Instance();
    returnClass.PublishSubscribe();

    theRFI.registerCreateMessage(dropAnchorInteraction->classHandleGet(), createDropAnchor);
    theRFI.registerCreateMessage(weighAnchorsInteraction->classHandleGet(), createWeighAnchors);
    theRFI.registerCreateMessage(setSailInteraction->classHandleGet(), createSetSail);

    Output::ready(wait);

    char str[128];
    sprintf(str, "Creating␣%d␣boats", nb_boats);
    wait = Output::wait(str);

    for (int i=0; i<nb_boats; i++) {
        char name[32];
        sprintf(name, "boat_%d", i);
        MyBoat* boat = new MyBoat(name);
        theRFI.registerObjectInstance(boat);
    }

    Output::ready(wait);

    for (int t=0; t<3000; t++) {
        RFI::processPendings();
        RFI::tick();
        // sleep for 100 milliseconds
        msSleep(100);
    }

    cout << "Done." << endl;

    return 0;

}
```

Listing B.4: Control a Boat Using Method Calls — controlBoat.cpp

```
// !!! WARNING !!!
// Please never add or remove any line from this file without updating the
// msf.tex documentation accordingly (Lorenzo)

#include "rfi.h"
#include "ftkfederateambassador.h"

#include "SailboatInstance.h"
#include "ReturnValueInstance.h"

#include "classes_hierarchy.h"

#include <stdio.h>  // sprintf...

using namespace msf;
```

```
int count_boats()
{
   static ftk::HANDLE boatHandle =
      (SingletonHolder<SailboatObjClass>::Instance()).classHandleGet();
   int n = 0;
   HLAObjectInstance* obj = RFI::getFirstObject();
   while ( obj ) {
      if ( obj->classHandleGet() == boatHandle ) n++;
      obj = RFI::getNextObject();
   }
   return n;
}

bool control_wait_for_acknowledgment(msf::U_LONG cmdID)
{
      bool acknowledge = false;
      long sec, msec;
      Timer timer;

      for (int t=0; t<1000 && !acknowledge; t++) {
         RFI::tick();
         if ( -1 != RFI::getReturnValue(cmdID) ) {
            acknowledge = true;
         }
         msSleep(5);
      }
      if ( acknowledge ) {
         timer.elapsed(sec, msec);
         cout << "  -> acknowledgment in "
              << sec << "s and " << msec << "ms" << endl;
         return true;
      }
      else {
         cout << "Time out for acknowledgment!" << endl;
         return false;
      }
}

HLAObjectInstance* createBoat(ftk::HANDLE objHandle, ftk::HANDLE classHandle)
{
   return new SailboatInstance(objHandle, classHandle);
}

HLAObjectInstance* createReturn(ftk::HANDLE objHandle, ftk::HANDLE classHandle)
{
   return new ReturnValueInstance(objHandle, classHandle);
}

int main (unsigned int argc, char *argv[])
{

   if ( argc < 2 ) {
      cerr << "Usage: " << argv[0] << " number_of_boats" << endl;
      exit(1);
```

```
}
int nb_boats = atoi(argv[1]);

RFI theRFI;
FTKFederateAmbassador* theFA = new FTKFederateAmbassador();
theRFI.federateAmbassadorSet(theFA);

theRFI.federationNameSet("MSF-TEST");
theRFI.federationFileSet("navigation.fed");

int wait = Output::wait("Trying␣to␣Join␣the␣Federation");
int error =  theRFI.joinFederation();
if ( error ) {
   Output::fatal("An␣Error␣as␣occured␣when␣joining:␣stop␣the␣program!\n");
}
else {
   Output::ready(wait);
}

wait = Output::wait("Getting␣all␣handles");
buildClassesHierarchy();
theRFI.getHandles();
Output::ready(wait);

wait = Output::wait("Publishing␣federates␣classes");
sailingBoatClass->unPublishAll();
sailingBoatClass->subscribeAll();
sailingBoatClass->PublishSubscribe();


dropAnchorInteraction->PublishSubscribe();
weighAnchorsInteraction->PublishSubscribe();
setSailInteraction->PublishSubscribe();

HLAObjectClass& returnClass = SingletonHolder<ReturnValueObjClass>::Instance();
returnClass.PublishSubscribe();

theRFI.registerCreateInstance(sailingBoatClass->classHandleGet(), createBoat);
theRFI.registerCreateInstance(returnClass.classHandleGet(), createReturn);

Output::ready(wait);

char str[128];
sprintf(str, "Waiting␣for␣%d␣boats", nb_boats);
wait = Output::wait(str);

for (int t=0; t<1000 &&  count_boats()<nb_boats; t++) {
   RFI::processPendings();
   RFI::tick();
   // sleep for 100 milliseconds
   msSleep(100);
}

if ( count_boats() == nb_boats ) {
   Output::ready(wait);
```

```
      cout << "Now␣set␣sails..." << endl;

      char name[32];
      char anchor[32];
      msf::U_LONG cmdID;

      for (int i=0; i<nb_boats; i++) {
         sprintf(name, "boat_%d", i);
         HLAObjectInstance* obj = theRFI.getObjectInstance(name);
         SailboatInstance* boat = dynamic_cast<SailboatInstance*>(obj);
         if ( boat ) {

            sprintf(anchor, "A_%c", char(i+65));
            cout << "Drop␣Anchor␣" << anchor << "␣with␣"
                 << (i+1)*10 << "␣meter␣of␣chain␣" << "␣of␣boat␣" << name << endl;
            cmdID = boat->dropAnchor(string(anchor), (float)(i+1)*10);
            control_wait_for_acknowledgment(cmdID);

            cout << "Weigh␣Anchors␣" << "␣of␣boat␣" << name << endl;
            cmdID = boat->weighAnchors();
            control_wait_for_acknowledgment(cmdID);

            cout << "Set␣sail␣" << i << "␣of␣boat␣" << name << endl;
            cmdID = boat->setSail(i);
            control_wait_for_acknowledgment(cmdID);

         }
      }

      cout << "Done." << endl;

   }
   else {
      Output::error("Time␣out:␣not␣enough␣boats!\n");
      Output::error("Terminating␣now.\n");
   }

   return 0;

}
```

Listing B.5: Discovering Specific Objects Using Shells — discoverSushi.cpp

```
// test program - discoverSushis.cpp
// Lorenzo Fluckiger - August 2001

// This example shows the usage of TObjShell which implements the concept
// of Shells. A Shell is an Object which listen to the simulation to
// discover a HLAObjectInstance with agiven name, and request attribute
// and/or ownership of attributes on discovery. The Shell becomes ready
// only when the instance has been discovered, the requested attributes
// have been updated and/or acquisitions acquired.
//
// This example work in pair with the createSushis program.
// This federate create two shells, one requiring an attribute update
// and this other requiring an attribute acquisition. The program
// displays the status of the shells.

#include "FTK/rfi.h"
#include "FTK/ftkfederateambassador.h"

#include "SushiObjClass.h"
#include "MakiObjClass.h"
#include "NigiriObjClass.h"

#include "MakiInstance.h"
#include "NigiriInstance.h"

#include "FTK/discoverpolicy.h"
#include "FTK/tobjshell.h"

// define how to create Nigiri Instances
HLAObjectInstance* createNigiri(ftk::HANDLE objHandle, ftk::HANDLE classHandle)
{
   return new msf::NigiriInstance(objHandle, classHandle);
}

// define how to create Maki Instances
HLAObjectInstance* createMaki(ftk::HANDLE objHandle, ftk::HANDLE classHandle)
{
   return new msf::MakiInstance(objHandle, classHandle);
}

int main (unsigned int argc, char *argv[])
{

   if (argc < 2) {
      cerr << "Usage:␣" << endl;
      cerr << argv[0] << "␣time_alive(in␣seconds)" << endl;
      exit(1);
   }
   int seconds = atoi(argv[1]);

//    ftk::Debug::debugLevelSet(ftk::Debug::DBG_BOMBASTIC);

   // Create a RTI-Federate-Interface (it could be static or dynamic object)
   RFI rfi;

   // Create a Federate Ambassador: it has to be a dynamic object since we
   // pass the pointer to the RFI, which will take care of deleting at
   // exit.  In most of the cases, the default FTKFederateAmbassador can
   // be used.
   FTKFederateAmbassador *fa = new FTKFederateAmbassador();
   // Assign the Federate Ambassador to the RFI
   rfi.federateAmbassadorSet(fa);

   // Define the Federation Name to wich this federate will participate
   rfi.federationNameSet("MSF");
   // Define the name of the federation file describing the federation
   rfi.federationFileSet("sushis.fed");
```

```
// Create Singletons of the Object Classes which will be used by this federate
HLAObjectClass& sushiClass = SingletonHolder<msf::SushiObjClass>::Instance();
HLAObjectClass& makiClass = SingletonHolder<msf::MakiObjClass>::Instance();
HLAObjectClass& nigiriClass = SingletonHolder<msf::NigiriObjClass>::Instance();

// Build the Object Class hierarchy by registering the classes to the
// RFI. Only the classes needed by this federate have to be register
// to the RFI, However, all the classes above them are also required in
// order for the RFI to build the full class hierarchy. For example
// this federate will only deal with Makis and Nigiris, but it is
// necessary to also register the Sushi class from which Nigiri and
// Maki are derived. However, the order of declaration is not
// important.
rfi.registerObjectClass(makiClass);
rfi.registerObjectClass(nigiriClass);
rfi.registerObjectClass(sushiClass);

// Tell this federate to join the federation
if ( rfi.joinFederation() ) {
  exit(1);
}

// Retrieve all the handles for Object Classes, Interaciton Classes,
// Attributes and Parameters
rfi.getHandles();

// Declare that this federate will publish and subscribe to Maki and
// Nigiri Classes.
// By default the each class has all its attributes set to publish and
// subscribe: unPublishAll could have been used for example to restrict
// this federate to only listen, and not create instances.
makiClass.PublishSubscribe();
nigiriClass.PublishSubscribe();

// Define the object factory: how to create object instances in
// function of class handle
rfi.registerCreateInstance(makiClass.classHandleGet(), createMaki);
rfi.registerCreateInstance(nigiriClass.classHandleGet(), createNigiri);

// Create a Policy for the discovery of Maki: for each new Maki
// discovered, this federate would like to get an update of the
// attribute "pieces".
DiscoverPolicy makiPolicy;
makiPolicy.addUpdateRequest(makiClass.getAttributeHandle(msf::MakiObjClass::PIECES));

// Create a Policy for the discovery of Nigir: for each new Nigiri
// discovered, this federate would like to take ownership of the
// attribute "pair". Note that a policy can be defined for more than
// one attribute and can combine update and ownership requests.
// Warning: the object representing a policy has to exist during the all
// time of the simulation: RFI use it for its shell management.
DiscoverPolicy nigiriPolicy;
nigiriPolicy.addOwnershipRequest(nigiriClass.getAttributeHandle(msf::NigiriObjClass::PAIR));
```

```
// Now declare two object instance shells: this shell are characterized
// by a type of object and a name identifying the object the federate
// would like to discover.
// The shell could be static or dynamic object: its is up to the federate developer
// to handle this objects properly
TObjShell<msf::MakiInstance> makiShell("kapa", makiPolicy);
TObjShell<msf::NigiriInstance>* nigiriShell =
   new TObjShell<msf::NigiriInstance>("maguro", nigiriPolicy);

// The created shell are registerd to the RFI in order for it to manage them
// regarding the events of discovery, update, acquisition.
rfi.registerObjShell(&makiShell);
rfi.registerObjShell(nigiriShell);

int newStatus;
int makiStatus = -1;
int nigiriStatus = -1;

// Enter in the simulation loop
for (int t=0; t<seconds*10; t++) {

   newStatus = makiShell.getStatus();
   if ( newStatus != makiStatus ) {
      if ( newStatus == GenericObjShell::READY ) {
         // The Shell is ready: the instance has been discovered
         // and an update for the attribute "pieces" has been
         // received. This federate can the safely display it
         msf::MakiInstance* obj = makiShell.instancePtr();
         cout << "Maki␣Instance␣[" << obj->nameGet()
              << "]␣READY:␣" << obj->piecesGet() << endl;
      }
      if ( newStatus == GenericObjShell::EMPTY ) {
         cout << "Maki␣Shell␣is␣EMPTY." << endl;
      }
      if ( newStatus == GenericObjShell::NOT_INIT ) {
         cout << "Maki␣Shell␣has␣instance␣but␣attributes␣are␣not␣initialized." << endl;
      }
      makiStatus = newStatus;
   }

   newStatus  = nigiriShell->getStatus();
   if ( newStatus != nigiriStatus ) {
      if ( newStatus == GenericObjShell::READY ) {
         // The Shell is ready: the instance has been discovered
         // and ownership of "pair" acquired. This federate can
         // then update the pair attribute!
         msf::NigiriInstance* obj = nigiriShell->instancePtr();
         obj->pairSet(true);
         cout << "Nigiri␣Instance␣[" << obj->nameGet()
              << "]␣READY:␣setting␣the␣pair␣value␣to␣true." << endl;
      }
      if ( newStatus == GenericObjShell::EMPTY ) {
         cout << "Nigiri␣Shell␣is␣EMPTY." << endl;
      }
      if ( newStatus == GenericObjShell::NOT_OWNED ) {
            cout << "Nigiri␣Shell␣has␣instance␣but␣attributes␣are␣not␣owned." << endl;
        }
        nigiriStatus = newStatus;
    }


    // Process all the pendings this federate has
    RFI::processPendings();
    // Let a chance to the FederateAmbassador to do its callbacks
    RFI::tick();

    // Go to sleep for 0.1 second to let the CPU do other work
    msf::msSleep(100);

  }

  delete(nigiriShell);

  return 0;

}
```